Generating Models of Infinite-State Communication Protocols using Regular Inference with Abstraction*

Fides Aarts[1], Bengt Jonsson[2], Johan Uijen[1†], and Frits Vaandrager[1]

[1] Institute for Computing and Information Sciences, Radboud University Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, the Netherlands
{f.aarts,f.vaandrager}@cs.ru.nl
[2] Department of Computer Systems, Uppsala University, Sweden
bengt@it.uu.se

Abstract. In order to facilitate model-based verification and validation, effort is underway to develop techniques for generating models of communication system components from observations of their external behavior. Most previous such work has employed regular inference techniques which generate modest-size finite-state models. They typically suppress parameters of messages, although these have a significant impact on control flow in many communication protocols. We present a framework, which adapts regular inference to include data parameters in messages and states for generating components with large or infinite message alphabets. A main idea is to adapt the framework of predicate abstraction, successfully used in formal verification. Since we are in a black-box setting, the abstraction must be supplied externally, using information about how the component manages data parameters. We have implemented our techniques by connecting the LearnLib tool for regular inference with the protocol simulator ns-2, and generated models of SIP and TCP components as implemented in ns-2.

1 Introduction

Model-based techniques for verification and validation of communication protocols and reactive systems, including model checking [22] and model-based testing [11] have witnessed drastic advances in the last decades, and several commercial tools that support model checking and/or model-based testing have become available (e.g., FDR2, Reactis, Conformiq Designer, Smartesting Certifylt, SeppMed MBTsuite, All4Tex MaTeLo, Axini Test Manager, and QuviQ). These techniques require formal state-machine models that specify the intended behavior of system components, which ideally should be developed during specification and design. However, the construction of such models typically requires

* Supported in part by EC Proj. 231167 (CONNECT), STW project 11763 Integrating Testing And Learning of Interface Automata (ITALIA), and EU FP7 grant no 214755 (QUASIMODO). A preliminary version of this paper appeared as [3].

† Current affiliation: Logica Nederland B.V., johan.uijen@logica.com.

significant manual effort, implying that in practice often models are not available, or become outdated as the system evolves. Automated support for constructing models of the behavior of implemented components would therefore be extremely useful, e.g., for regression testing, for replacing manual testing by model based testing, for producing models of standardized protocols, for analyzing whether an existing system is vulnerable to attacks, etc. Techniques, developed for program analysis, that construct models from source code (e.g., [8, 24]) are often of limited use, due to the presence of library modules, third-party components, etc., that make analysis of source code difficult. We therefore consider techniques for constructing state machine models from observations of the external behavior of a system.

The construction of models from observations of component behavior can be performed using regular inference (aka automata learning) techniques [7, 16, 30, 43]. This class of techniques is now receiving increasing attention in the testing and verification community, e.g., for regression testing of telecommunication systems [23, 28], for integration testing [31, 31, 21], security protocol testing [46], and for combining conformance testing and model checking [40, 20]. Algorithms for regular inference pose a number of *queries*, each of which observes the component's output in response to a certain sequence of inputs, and produce a minimal deterministic finite-state machine which conforms to the observations. If sufficiently many queries are asked, the produced machine will be a model of the observed component.

Since regular inference techniques are designed for finite-state models, previous applications to model generation have been limited to generating a moderate-size finite-state view of the system behavior, implying that the alphabet must be made finite, e.g., by suppressing parameters of messages. However, parameters have a significant impact on control flow in typical protocols: they can be sequence numbers, configuration parameters, agent and session identifiers, etc. The influence of data on control flow is taken into account by model-based test generation tools, such as ConformiQ Designer [27] and Spec Explorer [50, 17]. It is therefore important to extend inference techniques to handle message alphabets and state-spaces with structures containing data parameters with large domains.

In this paper, we present a general framework for generating models of protocol components with large or infinite structured message alphabets and state spaces. The framework is inspired by predicate abstraction [32, 14], which has been successful for extending finite-state model checking to large and infinite state spaces. In contrast to that work, however, we are now in a black-box setting, so we cannot derive the abstraction directly from the source code of a component. Instead, we use an externally supplied abstraction layer, which translates between a large or infinite message alphabet of the component to be modeled and a small finite alphabet of the regular inference algorithm. Via regular inference, a finite-state model of the abstracted interface is inferred. The abstraction can then be reversed to generate a faithful model of the component.

We describe how to construct a suitable abstraction from knowledge about which operators are sufficient to express guards and operations on data in a faithful model of the component. We have implemented our techniques by connecting the LearnLib tool for regular inference with the protocol simulator ns-2, which provides implementations of standard protocols. We have used it to generate models of the ns-2 implementations of entities in the SIP and TCP protocols.

Related Work. Regular inference techniques have been used for several tasks in verification and test generation, e.g., to create models of environment constraints with respect to which a component should be verified [15], for regression testing to create a specification and test suite [23, 28], to perform model checking without access to source code or formal models [20, 40], for program analysis [6], and for formal specification and verification [15]. Groz, Li, and Shahbaz [31, 45, 21] extend regular inference to Mealy machines with data values, for use in integration testing, but use only a finite set of the data values in the obtained model. In particular, they do not infer internal state variables. Shu and Lee [46] learn the behavior of security protocol implementations for a finite subset of input symbols, which can be extended in response to new information obtained in counterexamples. Lorenzoli, Mariani, and Pezzé infer models of software components that consider both sequence of method invocations and their associated data parameters [34, 33]. They infer a control structure of possible sequence of method invocations. In addition, each invocation is annotated with a precondition on method parameters, possibly also correlated with accessible system variables. They use a passive learning approach where the model is inferred from a given sample of traces, forming the control structure by an extension of the k-tails algorithm, and using Daikon [12] to infer relations on method parameters. Their setup is that of passive learning: we use an active learning approach where we assume that new queries may be supplied to the system; this is an added requirement but allows to generate a more informative sample by choosing the generated input. Furthermore, their work generates constraints that hold for the observed sample; they do not aim to infer functional relationships between input and output parameters, nor to infer how internal data variables of a component are managed. In previous work, we have considered extensions of regular inference to handle data parameters. In [9], we have considered extensions of regular regular inference for models with data parameters that are restricted to being boolean, using a technique with lazy refinement of guards. These techniques for maintaining guards have inspired the more general notion of abstractions on input symbols presented in the current paper. We have also proposed extensions of regular inference to handle infinite-state systems, in which parameters of messages and state variables are from an unbounded domain. For the special case that the domain admits only equality tests, efficient extensions of the L^* algorithm have been developed [26, 35, 1, 25] based on register automata [13]. We have also considered extensions to timers [19, 18], however with worst-case complexities that do not immediately suggest an efficient implementation. This paper proposes a general framework for incorporating a range of such data domains,

into which techniques specialized for different data domains can be incorporated, and which we have also evaluated on realistic protocol models.

Organization. Basic definitions of Mealy machines and regular inference are recalled in Section 2. Our new abstraction technique is presented in Section 3. Section 4 gives a symbolic representation of Mealy machines and abstractions. Section 5 describes how abstractions can be constructed in a systematic way. The application to SIP and TCP is reported in Section 6. Section 7 contains conclusions and directions for future work. Appendices A, B and C display the (abstract) models that we learned for the SIP and TCP protocols. Appendix D lists all the symbols used in this paper.

2 Inference of Mealy Machines

In this section, in order to fix notation and terminology, we recall the definition of a Mealy machine and the basic setup of regular inference in Angluin-style.

2.1 Mealy machines

We will use *Mealy machines* to model communication protocol entities. A *(nondeterministic) Mealy machine (MM)* is a tuple $\mathcal{M} = \langle I, O, Q, q_0, \rightarrow \rangle$, where

- I, O, and Q are nonempty sets of *input symbols*, *output symbols*, and *states*, respectively,
- $q_0 \in Q$ is the *initial state*, and
- $\rightarrow \subseteq Q \times I \times O \times Q$ is the *transition relation*.

We write $q \xrightarrow{i/o} q'$ if $(q, i, o, q') \in \rightarrow$, and $q \xrightarrow{i/o}$ if there exists a q' such that $q \xrightarrow{i/o} q'$. Mealy machines are assumed to be *input enabled* (or *completely specified*): for each state q and input i, there exists an output o such that $q \xrightarrow{i/o}$. An intuitive interpretation of a Mealy machine is as follows. At any point in time, the machine is in some state $q \in Q$. It is possible to give inputs to the machine by supplying an input symbol $i \in I$. The machine then (nondeterministically) selects a transition $q \xrightarrow{i/o} q'$, produces output symbol o, and jumps to the new state q'.

Mealy machine $\mathcal{M}$ is *deterministic* if for each state q and each input symbol i there is exactly one output symbol o and exactly one state q' such that $q \xrightarrow{i/o} q'$. We say that a Mealy machine is *finite* if the set Q of states and the set I of inputs are finite.

The transition relation is extended to finite sequences by defining $\stackrel{u/s}{\Rightarrow}$ to be the least relation that satisfies, for $q, q', q'' \in Q$, $u \in I^*$, $s \in O^*$, $i \in I$, and $o \in O$,

- $q \stackrel{\epsilon/\epsilon}{\Rightarrow} q$, and

– if $q \xrightarrow{i/o} q'$ and $q' \overset{u/s}{\Rightarrow} q''$ then $q \overset{i\ u/o\ s}{\Rightarrow} q''$.

Here we use ϵ to denote the empty sequence. We write $|s|$ to denote the length of a sequence s. Observe that $q \overset{u/s}{\Rightarrow} q'$ implies $|u| = |s|$. A state $q \in Q$ is called *reachable* if $q_0 \overset{u/s}{\Rightarrow} q$, for some u and s.

An *observation* over input symbols I and output symbols O is a pair $(u, s) \in I^* \times O^*$ such that sequences u and s have the same length. For $q \in Q$, we define $obs_{\mathcal{M}}(q)$, the set of observations of $\mathcal{M}$ from state q, by

$$obs_{\mathcal{M}}(q) = \{(u, s) \in I^* \times O^* \mid \exists q' : q \overset{u/s}{\Rightarrow} q'\}.$$

We write $obs_{\mathcal{M}}$ as a shorthand for $obs_{\mathcal{M}}(q_0)$. Note that, since Mealy machines are input enabled, $obs_{\mathcal{M}}(q)$ contains at least one pair (u, s), for each input sequence $u \in I^*$. We call $\mathcal{M}$ *behavior deterministic* if $obs_{\mathcal{M}}$ contains exactly one pair (u, s), for each $u \in I^*$. It is easy to see that a deterministic Mealy machine is behavior deterministic. Figure 1 gives an example of a behavior deterministic Mealy machine that is not deterministic.

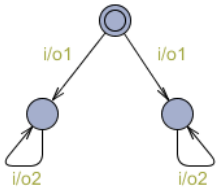

Fig. 1. Example Mealy machine that is behavior deterministic but not deterministic

Two states $q, q' \in Q$ are *observation equivalent*, denoted $q \approx q'$, if $obs_{\mathcal{M}}(q) = obs_{\mathcal{M}}(q')$. Two Mealy machines $\mathcal{M}_1$ and $\mathcal{M}_2$ with the same sets of input symbols are *observation equivalent*, notation $\mathcal{M}_1 \approx \mathcal{M}_2$, if $obs_{\mathcal{M}_1} = obs_{\mathcal{M}_2}$. We say that $\mathcal{M}_1$ *implements* $\mathcal{M}_2$, notation $\mathcal{M}_1 \leq \mathcal{M}_2$, if $\mathcal{M}_1$ and $\mathcal{M}_2$ have the same sets of input symbols and $obs_{\mathcal{M}_1} \subseteq obs_{\mathcal{M}_2}$.

The following lemma easily follows from the definitions.

Lemma 1. *If $\mathcal{M}_1 \leq \mathcal{M}_2$ and $\mathcal{M}_2$ is behavior deterministic then $\mathcal{M}_1 \approx \mathcal{M}_2$.*

We say that a Mealy machine is *finitary* if it is observation equivalent to a finite Mealy machine.

Example 1. An example of a finitary Mealy machine that is not finite is a Mealy machine with as states the set $\mathbb{N}$ of natural numbers, initial state 0, a single input i and a single output o, and transitions $n \xrightarrow{i/o} n+1$. This Mealy machine, which records in its state the number of inputs that has occurred, is equivalent to a Mealy machine with a single state q_0 and a single transition $q_0 \xrightarrow{i/o} q_0$.

2.2 Regular Inference

In order to learn Mealy machines, we define a slight generalization of the active learning setting of Angluin [7].

Let $\mathcal{M} = \langle I, O, Q, q_0, \rightarrow \rangle$ be a behavior deterministic Mealy machine. An *implementation* of $\mathcal{M}$ is a device that maintains the current state of $\mathcal{M}$, which at the beginning equals q_0. An implementation of $\mathcal{M}$ accepts inputs in I, called *output queries*, as well as a special reset input. Upon receiving query $i \in I$, the implementation picks a transition $q \xrightarrow{i/o} q'$, where q is its current state, generates output $o \in O$, and updates its current state to q'. Upon receiving a reset, the implementation resets its current state to q_0.

An *oracle* for $\mathcal{M}$ is a device which accepts an *inclusion query* or *hypothesis* $\mathcal{H}$ as input, where $\mathcal{H}$ is a Mealy machine with inputs I. Upon receiving an hypothesis $\mathcal{H}$, an oracle for $\mathcal{M}$ will produce output *yes* if the hypothesis is correct, that is, $\mathcal{M} \leq \mathcal{H}$, or else output a *counterexample*, which is an observation $(u, s) \in obs_{\mathcal{M}} - obs_{\mathcal{H}}$. The combination of an implementation of $\mathcal{M}$ and an oracle for $\mathcal{M}$ corresponds to what Angluin [7] calls a *teacher* for $\mathcal{M}$.

A *learner* for I is a device that may send inputs in $I \cup \{\mathsf{reset}\}$ to an implementation of $\mathcal{M}$, and Mealy machines $\mathcal{H}$ over I to an oracle for $\mathcal{M}$. The task of the learner is to learn a correct hypothesis in a finite number of steps, by observing the outputs generated by the implementation and the oracle in response to the queries.

Note that *inclusion queries* are slightly more general than the *equivalence queries* used by Angluin [7] and Niese [38]. However, if $\mathcal{M} \leq \mathcal{H}$ and moreover $\mathcal{H}$ is behavior deterministic then $\mathcal{M} \approx \mathcal{H}$ by Lemma 1. Hence, for a deterministic Mealy machine a hypothesis is correct in our setting iff it is correct in the settings of Angluin and Niese. The reason for our generalization will be discussed in Section 3. The typical behavior of a learner is to start by asking sequences of output queries (alternated with resets) until a "stable" hypothesis $\mathcal{H}$ can be built from the answers. After that an inclusion query is made to find out whether $\mathcal{H}$ is correct. If the answer is *yes* then the learner has succeeded. Otherwise the returned counterexample is used to perform subsequent output queries until converging to a new hypothesized automaton, which is supplied in an inclusion query, etc.

For finitary, behavior deterministic Mealy machines the above problem is well understood. The L^* algorithm, which has been adapted to Mealy machines by Niese [38], generates deterministic hypotheses $\mathcal{H}$ that are the minimal Mealy machines that agree with a performed set of output queries. Since in practice there is no oracle that can answer equivalence or inclusion queries, LearnLib "approximates" such queries by generating long test sequences that are computed using standard methods like the W-method. The algorithms have been implemented in the LearnLib tool [42, 36], developed at the Technical University of Dortmund.

3 Inference Using Abstraction

Existing implementations of inference algorithms only proved effective when applied to machines with small alphabets (sets of input and output symbols). Practical systems, however, typically have large alphabets, e.g. inputs and outputs with data parameters of type integer or string. In order to infer large- or infinite-state Mealy machines, we adapt ideas from predicate abstraction [32, 14], which have been successful for extending finite-state model checking to large and infinite state spaces. The main idea is to divide the concrete input domain into a small number of abstract equivalence classes in a history-dependent manner.

Example 2 (Component of a simple communication protocol). Consider a Mealy machine $\mathcal{M}_{COM}$ that models a component of a simple communication protocol. The component accepts request messages, which are modeled as inputs of $\mathcal{M}_{COM}$, and generates OK/NOK reply messages, which correspond to outputs of $\mathcal{M}_{COM}$. The set of inputs is $I = \{REQ(id, sn) \mid id, sn \in \mathbb{N}\}$, where parameter id is an identifier and parameter sn is a sequence number. The set of outputs is $O = \{OK, NOK\}$. The set of states is given by $Q = \mathbb{N} \times \mathbb{N} \times \mathbb{B}$, where the two natural numbers record the current values of id and sn, respectively, and the boolean value denotes whether the component has been initialized. The initial state is $q_0 = (0, 0, \mathsf{F})$. The transition relation contains the following transitions, for all $id, sn, id', sn' \in \mathbb{N}$,

$$
\begin{aligned}
&(id, sn, \mathsf{F}) \xrightarrow{REQ(id',sn')/OK} (id', sn', \mathsf{T}),\\
&(id, sn, \mathsf{T}) \xrightarrow{REQ(id',sn')/OK} (id', sn', \mathsf{T}) \text{ if } id' = id \text{ and } sn' = sn + 1, \text{ and}\\
&(id, sn, \mathsf{T}) \xrightarrow{REQ(id',sn')/NOK} (id, sn, \mathsf{T}) \text{ otherwise}
\end{aligned}
$$

With the first transition the "current" session is initialized by storing the id and sn received in the request message. If in any subsequent request the id of the "current" session is used in combination with the successor of the sequence number sn, an OK output is produced, otherwise a NOK output is returned.

Since all combinations of concrete values need to be inferred, e.g. $REQ(0, 0)$, $REQ(1, 0)$, and $REQ(1, 1)$, application of the L^* algorithm is impossible. To infer the machine, we place a mapper module in between the learner and the implementation that abstracts the set of concrete parameter values to (small) finite sets of abstract values, see Figure 2.

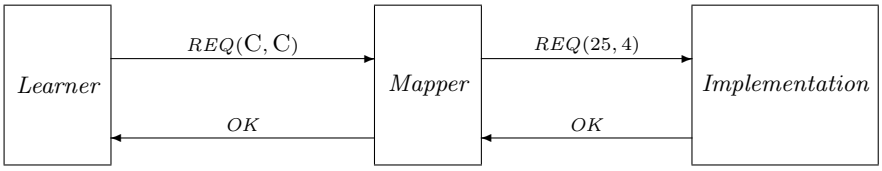

Fig. 2. Introduction of mapper component

Concrete symbols of form $REQ(id, sn)$ are abstracted to symbols of form $REQ(\mathrm{ID}, \mathrm{SN})$, where ID and SN are from a small domain, say $\{\mathrm{C}, \mathrm{O}\}$. We abstract the parameter value id by C if id is the identifier of the "current" session, and by O otherwise. We abstract the parameter sn in a similar way: to C if it is the successor of the current sequence number, and to O otherwise. Thus, for instance, input string $REQ(25, 4)$ $REQ(25, 7)$ is abstracted to $REQ(\mathrm{C}, \mathrm{C})$ $REQ(\mathrm{C}, \mathrm{O})$, whereas the input string $REQ(25, 4)$ $REQ(42, 5)$ is abstracted to $REQ(\mathrm{C}, \mathrm{C})$ $REQ(\mathrm{O}, \mathrm{C})$. The resulting abstraction is not "state-free", as it depends on the current values of the session. The mapper records these values in its state. □

In general, in order to learn an over-approximation of a "large" Mealy machine $\mathcal{M}$, we place a mapper in between the implementation and the learner, which translates the concrete inputs in I to the abstract inputs in X, the concrete outputs in O to the abstract outputs in Y, and vice versa. This will allow us to reduce the task of the learner to infering a "small" Mealy machine with alphabet X and Y. The next subsection formalizes the concept of a mapper and establishes some technical lemmas. After that, in Subsection 3.2, we show how we can turn the abstract model that the learner learns in the setup of Figure 2, into a correct model for the Mealy machine of the implementation.

3.1 Mappers

The behavior of the intermediate component is fully determined by the notion of a *mapper*. A mapper encompasses both concrete and abstract sets of input and output symbols, a set of states and a transition function that tells us how the occurrence of a concrete symbol affects the state, and an abstraction function which, depending on the state, maps concrete to abstract symbols.

Definition 1 (Mapper). *A* mapper *for a set of inputs I and a set of outputs O is a tuple $\mathcal{A} = \langle I, O, R, r_0, \delta, X, Y, \psi \rangle$, where*

- *I and O are disjoint sets of* concrete input and output symbols*,*
- *R is a set of* mapper states*,*
- *$r_0 \in R$ is an* initial mapper state*,*
- *$\delta : R \times (I \cup O) \to R$ is an* update function*; we write $r \xrightarrow{a} r'$ if $\delta(r, a) = r'$,*
- *X and Y are finite sets of* abstract input and output symbols*, and*
- *$\psi : R \times (I \cup O) \to (X \cup Y)$ is an* abstraction function *that respects inputs and outputs, that is, for all $a \in I \cup O$ and $r \in R$, $a \in I \Leftrightarrow \psi(r, a) \in X$.*

Example 3 (A mapper for $\mathcal{M}_{COM}$). We define $\mathcal{A} = \langle I, O, R, r_0, \delta, X, Y, \psi \rangle$, a mapper for the Mealy machine $\mathcal{M}_{COM}$ of Example 2. The sets I and O of the mapper are the same as for $\mathcal{M}_{COM}$. The mapper records the current values of id and sn in its state: $R = \{\bot\} \cup (\mathbb{N} \times \mathbb{N})$. Initially, no values for id and sn have been selected: $r_0 = \bot$. The state of the mapper only changes when a $REQ(id, sn)$

input arrives in the initial state, or when id is the current session identifier and sn the successor of the current sequence number:

$$\begin{aligned}\delta(\bot, REQ(id, sn)) &= (id, sn)\\ \delta((id, sn), REQ(id', sn')) &= (id', sn') \text{ if } id' = id \wedge sn' = sn + 1\\ \delta((id, sn), REQ(id', sn')) &= (id, sn) \text{ if } id' \neq id \vee sn' \neq sn + 1\end{aligned}$$

Output actions do not change the state of the mapper: $\delta(r, o) = r$, for $r \in R$ and $o \in O$. The set of abstract input symbols is

$$X = \{REQ(\mathrm{C}, \mathrm{C}), REQ(\mathrm{C}, \mathrm{O}), REQ(\mathrm{O}, \mathrm{C}), REQ(\mathrm{O}, \mathrm{O})\},$$

and the set of abstract output symbols Y equals the set of concrete outputs O. In the initial state the abtraction function maps all parameter values to C:

$$\psi(\bot, REQ(id, sn)) = REQ(\mathrm{C}, \mathrm{C}).$$

The abstraction function forgets the concrete parameter values of any subsequent request and only records whether they are correct or not:

$$\psi((id, sn), REQ(id', sn')) = REQ(\mathrm{ID}, \mathrm{SN}),$$

where ID = **if** $id' = id$ **then** C **else** O, and SN = **if** $sn' = sn + 1$ **then** C **else** O. For outputs ψ acts as the identity function. □

A mapper allows us to abstract a Mealy machine with concrete symbols in I and O into a Mealy machine with abstract symbols in X and Y, and, conversely, to concretize a Mealy machine with symbols in X and Y into a Mealy machine with symbols in I and O. First we show how an abstract Mealy machine can be built from a mapper and a concrete Mealy machine, and explore some properties of this construction. Basically, the abstraction of Mealy machine $\mathcal{M}$ via mapper $\mathcal{A}$ is the Cartesian product of the underlying transition systems, in which the abstraction function is used to convert concrete symbols into abstract ones.

Definition 2 (Abstraction). *Let $\mathcal{M} = \langle I, O, Q, q_0, \rightarrow\rangle$ be a Mealy machine and let $\mathcal{A} = \langle I, O, R, r_0, \delta, X, Y, \psi\rangle$ be a mapper. Then $\alpha_{\mathcal{A}}(\mathcal{M})$, the* abstraction of $\mathcal{M}$ via $\mathcal{A}$*, is the Mealy machine $\langle X, Y \cup \{\bot\}, Q \times R, (q_0, r_0), \rightarrow'\rangle$, where $\rightarrow'$ is given by the rules*

$$\frac{q \xrightarrow{i/o} q',\ r \xrightarrow{i} r' \xrightarrow{o} r'',\ \psi(r, i) = x,\ \psi(r', o) = y}{(q, r) \xrightarrow{x/y}{}' (q', r'')} \qquad \frac{\nexists i : \psi(r, i) = x}{(q, r) \xrightarrow{x/\bot}{}' (q, r)}$$

The second rule in the definition is required to ensure that the abstraction $\alpha_{\mathcal{A}}(\mathcal{M})$ is input enabled. Given a state of the mapper, it may occur that for some abstract input symbol x there is no corresponding concrete input symbol i. In this case, an input x triggers a special "undefined" output symbol $\bot$ and leaves the state unchanged.

Example 4 (Abstraction of $\mathcal{M}_{COM}$). The abstraction $\alpha_{\mathcal{A}}(\mathcal{M}_{COM})$ of our example Mealy machine $\mathcal{M}_{COM}$ has the same abstract input and output symbols as mapper $\mathcal{A}$, except that there is an additional "undefined" abstract output symbol $\perp$. States of the abstract Mealy machine $\alpha_{\mathcal{A}}(\mathcal{M}_{COM})$ are pairs (q, r) where q is a state of Mealy machine $\mathcal{M}_{COM}$ and r is a state of mapper $\mathcal{A}$. The initial state is $((0, 0, \mathsf{F}), \perp)$. We have the following transitions, for all $sn, id \in \mathbb{N}$ (only transitions reachable from the initial state are listed):

$$
\begin{aligned}
((0,0,\mathsf{F}),\perp) &\xrightarrow{REQ(\mathrm{C},\mathrm{C})/OK}{}' ((id,sn,\mathsf{T}),(id,sn))\\
((0,0,\mathsf{F}),\perp) &\xrightarrow{REQ(\mathrm{C},\mathrm{O})/\perp}{}' ((0,0,\mathsf{F}),\perp)\\
((0,0,\mathsf{F}),\perp) &\xrightarrow{REQ(\mathrm{O},\mathrm{C})/\perp}{}' ((0,0,\mathsf{F}),\perp)\\
((0,0,\mathsf{F}),\perp) &\xrightarrow{REQ(\mathrm{O},\mathrm{O})/\perp}{}' ((0,0,\mathsf{F}),\perp)\\
((id,sn,\mathsf{T}),(id,sn)) &\xrightarrow{REQ(\mathrm{C},\mathrm{C})/OK}{}' ((id,sn+1,\mathsf{T}),(id,sn+1))\\
((id,sn,\mathsf{T}),(id,sn)) &\xrightarrow{REQ(\mathrm{C},\mathrm{O})/NOK}{}' ((id,sn,\mathsf{T}),(id,sn))\\
((id,sn,\mathsf{T}),(id,sn)) &\xrightarrow{REQ(\mathrm{O},\mathrm{C})/NOK}{}' ((id,sn,\mathsf{T}),(id,sn))\\
((id,sn,\mathsf{T}),(id,sn)) &\xrightarrow{REQ(\mathrm{O},\mathrm{O})/NOK}{}' ((id,sn,\mathsf{T}),(id,sn))
\end{aligned}
$$

Observe that, by the second rule in Definition 2, the abstract inputs $REQ(\mathrm{C}, \mathrm{O})$, $REQ(\mathrm{O}, \mathrm{C})$, and $REQ(\mathrm{O}, \mathrm{O})$ in the initial state trigger an output $\perp$, since in this state all concrete input actions are mapped to $REQ(\mathrm{C}, \mathrm{C})$. It is not hard to see

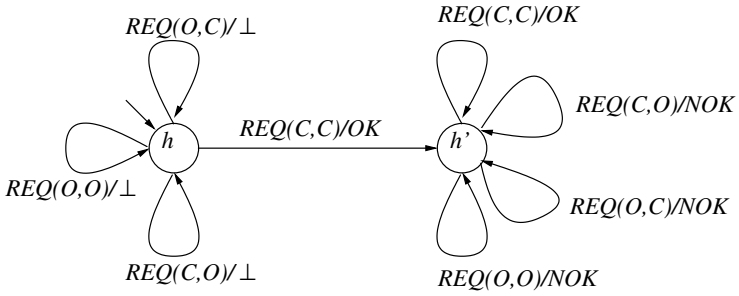

Fig. 3. Minimal Mealy machine $\mathcal{H}_{COM}$ equivalent to $\alpha_{\mathcal{A}}(\mathcal{M}_{COM})$

that $\alpha_{\mathcal{A}}(\mathcal{M}_{COM})$ is observation equivalent to the deterministic Mealy machine $\mathcal{H}_{COM}$ displayed in Figure 3. □

The abstraction function of a mapper can be lifted to observations in a straightforward manner: every concrete input or output string can be turned into an abstract string by stepwise transforming every symbol according to ψ:

Definition 3 (Abstraction of observations). *Let $\mathcal{A}$ be a mapper. Then function $\tau_{\mathcal{A}}$, which maps concrete observations over I and O to abstract observations*

over X and Y, is defined inductively by

$$\tau_{\mathcal{A}}(u,s) = \tau_{\mathcal{A}}(u,s,r_0) \tag{1}$$
$$\tau_{\mathcal{A}}(u,s,r) = (\tau^I_{\mathcal{A}}(u,s,r), \tau^O_{\mathcal{A}}(u,s,r)) \tag{2}$$
$$\tau^I_{\mathcal{A}}(\epsilon,\epsilon,r) = \epsilon \tag{3}$$
$$\tau^O_{\mathcal{A}}(\epsilon,\epsilon,r) = \epsilon \tag{4}$$
$$\tau^I_{\mathcal{A}}(iu,os,r) = \psi(r,i)\ \tau^I_{\mathcal{A}}(u,s,r'') \tag{5}$$
$$\tau^O_{\mathcal{A}}(iu,os,r) = \psi(r',o)\ \tau^O_{\mathcal{A}}(u,s,r'') \tag{6}$$

where $r' = \delta(r,i)$ and $r'' = \delta(r',o)$.

For a given mapper $\mathcal{A}$, the abstraction operator on Mealy machines is of course closely related to the abstraction operator on observations. The connection is formally established in Claim 1 below. Using the claim, we link observations of $\alpha_{\mathcal{A}}(\mathcal{M})$ to observations of $\mathcal{M}$ in Lemma 2. First, we need some notation. Update function δ is extended to a function from $R \times (I \cup O)^* \to R$ by

$$\delta(r,\epsilon) = r \tag{7}$$
$$\delta(r,a\ u) = \delta(\delta(r,a),u) \tag{8}$$

We write $r \overset{u/s}{\rightsquigarrow} r'$ iff $r' = \delta(r, zip(u,s))$, where zip is defined as follows:

$$zip(\epsilon,\epsilon) = \epsilon \tag{9}$$
$$zip(i\ u, o\ s) = i\ o\ zip(u,s) \tag{10}$$

Claim 1 *Suppose $q \overset{u/s}{\Rightarrow} q'$ and $r \overset{u/s}{\rightsquigarrow} r'$. Then $(q,r) \overset{\tau^I_{\mathcal{A}}(u,s,r)/\tau^O_{\mathcal{A}}(u,s,r)}{\Rightarrow} (q',r')$.*

Proof. By induction on the length of u. Let $u' = \tau^I_{\mathcal{A}}(u,s,r)$ and $s' = \tau^O_{\mathcal{A}}(u,s,r)$. Basis: $|u| = 0$. Then $u = \epsilon$ and because $q \overset{u/s}{\Rightarrow} q'$ implies $|u| = |s|$, also $s = \epsilon$. By the inductive definition of $\overset{u/s}{\Rightarrow}$ and $q \overset{\epsilon/\epsilon}{\Rightarrow} q'$, it follows that $q = q'$. Furthermore, $r \overset{\epsilon/\epsilon}{\rightsquigarrow} r'$ implies $r' = \delta(r, zip(\epsilon,\epsilon))$, which in turn implies $r' = r$ by Equations (9) and (7). This implies $(q,r) \overset{\tau^I_{\mathcal{A}}(u,s,r)/\tau^O_{\mathcal{A}}(u,s,r)}{\Rightarrow} (q',r')$, by Equations (3) and (4), and by the inductive definition of $\overset{u/s}{\Rightarrow}$, as required.
Induction step: Assume $u = i\bar{u}$, where $i \in I$ and $\bar{u}$ is of length n. Then we can

write $s = o\bar{s}$, where $o \in O$ and $\bar{s}$ is of length n. We infer

$$
\begin{aligned}
q \overset{u/s}{\Rightarrow} q' \wedge r \overset{u/s}{\rightsquigarrow} r' &\Rightarrow \left(\begin{smallmatrix}\text{Assumption}\\ \text{on u and s}\end{smallmatrix}\right)\\
q \overset{i\bar{u}/o\bar{s}}{\Rightarrow} q' \wedge r \overset{i\bar{u}/o\bar{s}}{\rightsquigarrow} r' &\Rightarrow \left(\begin{smallmatrix}\text{Definition}\\ \overset{u/s}{\Rightarrow}\end{smallmatrix}\right)\\
\exists\, q'' : q \xrightarrow{i/o} q'' \wedge q'' \overset{\bar{u}/\bar{s}}{\Rightarrow} q' \wedge r \overset{i\bar{u}/o\bar{s}}{\rightsquigarrow} r' &\Rightarrow \left(\begin{smallmatrix}\text{Definition}\\ \delta \text{ and } \mathit{zip}\end{smallmatrix}\right)\\
\exists\, q''\exists\, r_1, r_2 : q \xrightarrow{i/o} q'' \wedge q'' \overset{\bar{u}/\bar{s}}{\Rightarrow} q' \wedge r \xrightarrow{i} r_1 \wedge r_1 \xrightarrow{o} r_2 \wedge r_2 \overset{\bar{u}/\bar{s}}{\rightsquigarrow} r' &\Rightarrow \left(\begin{smallmatrix}\text{Inductive}\\ \text{hypothesis}\end{smallmatrix}\right)\\
\exists\, q''\exists r_1, r_2 :\\
q \xrightarrow{i/o} q'' \wedge r \xrightarrow{i} r_1 \wedge r_1 \xrightarrow{o} r_2 \wedge (q'', r_2) \overset{\tau^I_{\mathcal{A}}(\bar{u},\bar{s},r_2)/\tau^O_{\mathcal{A}}(\bar{u},\bar{s},r_2)}{\Rightarrow} (q', r') &\Rightarrow \begin{pmatrix}\text{Definition}\\ \alpha_{\mathcal{A}}(\mathcal{M})\end{pmatrix}\\
\exists\, q''\exists r_1, r_2 : r \xrightarrow{i} r_1 \wedge r_1 \xrightarrow{o} r_2 \wedge\\
(q, r) \xrightarrow{\psi(r,i)/\psi(r_1,o)}{}' (q'', r_2) \wedge (q'', r_2) \overset{\tau^I_{\mathcal{A}}(\bar{u},\bar{s},r_2)/\tau^O_{\mathcal{A}}(\bar{u},\bar{s},r_2)}{\Rightarrow} (q', r') &\Rightarrow \begin{pmatrix}\text{Definition}\\ \overset{u/s}{\Rightarrow}\end{pmatrix}\\
\exists r_1, r_2 : r \xrightarrow{i} r_1 \wedge r_1 \xrightarrow{o} r_2 \wedge\\
(q, r) \overset{\psi(r,i)\tau^I_{\mathcal{A}}(\bar{u},\bar{s},r_2)/\psi(r_1,o)\tau^O_{\mathcal{A}}(\bar{u},\bar{s},r_2)}{\Rightarrow} (q', r') &\Rightarrow \begin{pmatrix}\text{Equations}\\ (5) \text{ and } (6)\end{pmatrix}\\
(q, r) \overset{\tau^I_{\mathcal{A}}(i\bar{u},o\bar{s},r),\tau^O_{\mathcal{A}}(i\bar{u},o\bar{s},r)}{\Rightarrow} (q', r')
\end{aligned}
$$

Hence, $q \overset{u/s}{\Rightarrow} q'$ and $r \overset{u/s}{\rightsquigarrow} r'$ implies $(q, r) \overset{\tau^I_{\mathcal{A}}(u,s,r)/\tau^O_{\mathcal{A}}(u,s,r)}{\Rightarrow} (q', r')$, as required.

Lemma 2. *Suppose* $(u, s) \in obs_{\mathcal{M}}$. *Then* $\tau_{\mathcal{A}}(u, s) \in obs_{\alpha_{\mathcal{A}}(\mathcal{M})}$.

Proof. Let r' be the unique mapper state such that $r_0 \overset{u/s}{\rightsquigarrow} r'$. Then

$$
\begin{aligned}
(u, s) \in obs_{\mathcal{M}} &\Rightarrow \text{ (Definition of } obs)\\
\exists q' : q_0 \overset{u/s}{\Rightarrow} q' &\Rightarrow \text{ (Claim 1)}\\
\exists q' : (q_0, r_0) \overset{\tau^I_{\mathcal{A}}(u,s,r_0)/\tau^O_{\mathcal{A}}(u,s,r_0)}{\Rightarrow} (q', r') &\Rightarrow \text{ (Definition of } obs)\\
(\tau^I_{\mathcal{A}}(u, s, r_0), \tau^O_{\mathcal{A}}(u, s, r_0)) \in obs_{\alpha_{\mathcal{A}}(\mathcal{M})} &\Rightarrow \text{ (Equations (1) and (2))}\\
\tau_{\mathcal{A}}(u, s) \in obs_{\alpha_{\mathcal{A}}(\mathcal{M})}
\end{aligned}
$$

We now define the *concretization operator*, which is the adjoint of the abstraction operator. For a given mapper $\mathcal{A}$, the corresponding concretization operator turns any abstract Mealy machine with symbols in X and Y into a concrete Mealy machine with symbols in I and O. Basically, the concretization of Mealy machine $\mathcal{H}$ via mapper $\mathcal{A}$ is the Cartesian product of the underlying transition systems, in which the abstraction function is used to convert abstract symbols into concrete ones.

Definition 4 (Concretization). *Let* $\mathcal{H} = \langle X, Y \cup \{\bot\}, H, h_0, \rightarrow\rangle$ *be a Mealy machine and let* $\mathcal{A} = \langle I, O, R, r_0, \delta, X, Y, \psi\rangle$ *be a mapper for* I *and* O. *Then*

$\gamma_{\mathcal{A}}(\mathcal{H})$, the concretization *of $\mathcal{H}$ via $\mathcal{A}$, is the Mealy machine $\langle I, O \cup \{\bot\}, R \times H, (r_0, h_0), \rightarrow'' \rangle$, where $\rightarrow''$ is given by the rules*

$$\frac{r \xrightarrow{i} r' \xrightarrow{o} r'',\ \psi(r,i) = x,\ \psi(r',o) = y,\ h \xrightarrow{x/y} h'}{(r,h) \xrightarrow{i/o}'' (r'',h')}$$

$$\frac{r \xrightarrow{i} r',\ \psi(r,i) = x,\ h \xrightarrow{x/y} h',\quad \nexists o \in O : \psi(r',o) = y}{(r,h) \xrightarrow{i/\bot}'' (r,h)}$$

States of the concretization $\gamma_{\mathcal{A}}(\mathcal{H})$ are pairs (r,h) of a state h of the hypothesis and a state r of the mapper. Each transition $h \xrightarrow{x/y} h'$ of the hypothesis corresponds to potentially many transitions of the concretization: (r,h) has an outgoing i/o transition whenever $\psi(r,i) = x$ and $\psi(r',o) = y$, where r' is the unique state such that $r \xrightarrow{i} r'$. The second rule in the definition is required to ensure the concretization $\gamma_{\mathcal{A}}(\mathcal{H})$ is input enabled. Consider a state (r,h) of the concretization and a concrete input i. Since $\mathcal{A}$ is deterministic and input enabled, there exists a unique state r' such that $r \xrightarrow{i} r'$. Let $x = \psi(r,i)$ be the corresponding abstract input. Since $\mathcal{H}$ is input enabled, there also exists a state h' and an abstract output y such that $h \xrightarrow{x/y} h'$. However, there does not necessarily exist an output o with $\psi(r',o) = y$. This means that the first rule cannot always be applied to infer an outgoing i-transition of state (r,h). In order to ensure input enabledness, the second rule is used in this case to introduce a transition with "undefined" output $\bot$ that leaves the state (r,h) unchanged.

Example 5 (Concretization of $\mathcal{H}_{COM}$). Let us now concretize the abstract Mealy machine $\mathcal{H}_{COM}$ of Figure 3, which is observation equivalent to $\alpha_{\mathcal{A}}(\mathcal{M}_{COM})$. The Mealy machine $\gamma_{\mathcal{A}}(\mathcal{H}_{COM})$ has the same concrete input and output symbols as $\mathcal{M}_{COM}$, except for the additional output $\bot$. States of the concretization are pairs of states of $\mathcal{A}$ and states of $\mathcal{H}_{COM}$. The initial state is $(\bot,h)$. We have the following transitions, for all $id, id', sn, sn' \in \mathbb{N}$ with $id' \neq id$ and $sn' \neq sn+1$ (only transitions reachable from the initial state are listed):

$$\begin{aligned}
(\bot,h) &\xrightarrow{REQ(id,sn)/OK}' ((id,sn),h') \\
((id,sn),h') &\xrightarrow{REQ(id,sn+1)/OK}' ((id,sn+1),h') \\
((id,sn),h') &\xrightarrow{REQ(id,sn')/NOK}' ((id,sn),h') \\
((id,sn),h') &\xrightarrow{REQ(id',sn+1)/NOK}' ((id,sn),h') \\
((id,sn),h') &\xrightarrow{REQ(id',sn')/NOK}' ((id,sn),h')
\end{aligned}$$

Note that the transitions with output $\bot$ in $\mathcal{H}_{COM}$ play no role in $\gamma_{\mathcal{A}}(\mathcal{H}_{COM})$ since there exists no concrete output of $\mathcal{A}$ that is abstracted to $\bot$: the only use of these transitions is to make $\mathcal{H}_{COM}$ input enabled. Also note that in this specific example the second rule of Definition 4 does not play a role, since ψ acts

as the identity function on outputs. The reader may check that $\gamma_{\mathcal{A}}(\mathcal{H}_{COM})$ is observation equivalent to $\mathcal{M}_{COM}$. □

Claim 2 and Lemma 3 below link the behavior of the concretization $\gamma_{\mathcal{A}}(\mathcal{H})$ to the behavior of $\mathcal{H}$.

Claim 2 *Let (u, s) be an observation over inputs I and outputs O. Suppose $r \overset{u/s}{\rightsquigarrow} r'$ and $h \overset{\tau^I_{\mathcal{A}}(u,s,r)/\tau^O_{\mathcal{A}}(u,s,r)}{\Rightarrow} h'$. Then $(r, h) \overset{u/s}{\Rightarrow} (r', h')$ (where $\Rightarrow$ is the extension of the transition relation of $\gamma_{\mathcal{A}}(\mathcal{H})$ to finite sequences).*

Proof. Proof by induction on length of u. Let $u' = \tau^I_{\mathcal{A}}(u, s, r)$ and $s' = \tau^O_{\mathcal{A}}(u, s, r)$. Basis: $|u| = 0$. Then $u = \epsilon$ and, because $q \overset{u/s}{\Rightarrow} q'$ implies $|u| = |s|$, also $s = \epsilon$. According to Equations 3 and 4, $u' = \tau^I_{\mathcal{A}}(\epsilon, \epsilon, r) = \epsilon$ and $s' = \tau^O_{\mathcal{A}}(\epsilon, \epsilon, r) = \epsilon$. By the definition of $\overset{u/s}{\Rightarrow}$ and $h \overset{\epsilon/\epsilon}{\Rightarrow} h'$ it follows that $h = h'$. Hence, $h = h'$. Moreover, $r \overset{\epsilon/\epsilon}{\rightsquigarrow} r'$, where $zip(\epsilon, \epsilon) = \epsilon$ implies $\delta(r, \epsilon) = r$ and thus $r = r'$, see transition relation δ in Definition 1. Hence $(r, h) \overset{\epsilon/\epsilon}{\Rightarrow} (r', h')$ and thus $(r, h) \overset{u/s}{\Rightarrow} (r', h')$, as required.

Induction step: Assume $u = i\bar{u}$, where $\bar{u}$ is of length n. Then we can write $s = o\bar{s}$, where $\bar{s}$ is of length n. By combining the above observations and by Equations 5 and 6 we infer:
$(\tau^I_{\mathcal{A}}(i\bar{u}, o\bar{s}, r), \tau^O_{\mathcal{A}}(i\bar{u}, o\bar{s}, r)) = (\psi(r, i)\ \tau^I_{\mathcal{A}}(\bar{u}, \bar{s}, r_2), \psi(r_1, o)\ \tau^O_{\mathcal{A}}(\bar{u}, \bar{s}, r_2))$,
where $r_2 = \delta(r_1, o)$ and $r_1 = \delta(r, i)$.
Let $i' = \psi(r, i)$, $\bar{u}' = \tau^I_{\mathcal{A}}(\bar{u}, \bar{s}, r_2)$, $o' = \psi(r_1, o)$ and $\bar{s}' = \tau^O_{\mathcal{A}}(\bar{u}, \bar{s}, r_2)$. Then $u' = i'\bar{u}'$ and $s' = o'\bar{s}'$ and $h \overset{i'\bar{u}'/o'\bar{s}'}{\Rightarrow} h'$. Hence, by definition of $\overset{u/s}{\Rightarrow}$ there exists a h'' such that $h \overset{i'/o'}{\rightarrow} h''$ and $h'' \overset{\bar{u}'/\bar{s}'}{\Rightarrow} h'$, which is $\boxed{h'' \overset{\tau^I_{\mathcal{A}}(\bar{u},\bar{s},r_2)/\tau^O_{\mathcal{A}}(\bar{u},\bar{s},r_2)}{\Rightarrow} h'}$. We infer

$$
\begin{array}{lll}
 & r \overset{u/s}{\rightsquigarrow} r' & \text{(Definition of } u \text{ and } s\text{)} \\
\Leftrightarrow & r \overset{i\bar{u}/o\bar{s}}{\rightsquigarrow} r' & \text{(Definition of } \delta\text{)} \\
\Leftrightarrow & r' = \delta(r, zip(i\bar{u}, o\bar{s})) & \text{(Definition of } zip\text{)} \\
\Leftrightarrow & r' = \delta(r, i\ o\ zip(\bar{u}, \bar{s})) & \text{(Definition of } r_1 \text{ and } r_2\text{)} \\
\Leftrightarrow & r' = \delta(r_2, zip(\bar{u}, \bar{s})) & \text{(Definition of } \delta\text{)} \\
\Leftrightarrow & \boxed{r_2 \overset{\bar{u}/\bar{s}}{\rightsquigarrow} r'} &
\end{array}
$$

By the induction hypothesis, the combination of the two boxed assertions implies $(r_2, h'') \overset{\bar{u}/\bar{s}}{\Rightarrow} (r', h')$. By definition of r_1 and r_2, $r \overset{i}{\rightarrow} r_1 \overset{o}{\rightarrow} r_2$. Since moreover $h \xrightarrow{\psi(r,i)/\psi(r',o)} h''$, application of the transition rule for $\gamma_{\mathcal{A}}(\mathcal{H})$ gives $(r, h) \xrightarrow{i/o} {}''(r_2, h'')$. Combination with $(r_2, h'') \overset{\bar{u}/\bar{s}}{\Rightarrow} (r', h')$ yields $(r, h) \overset{i\bar{u}/o\bar{s}}{\Rightarrow} (r', h')$. Hence, $(r, h) \overset{u/s}{\Rightarrow} (r', h')$, as required.

Lemma 3. *Let (u, s) be an observation over inputs I and outputs O. Then $\tau_{\mathcal{A}}(u, s) \in obs_{\mathcal{H}}$ implies $(u, s) \in obs_{\gamma_{\mathcal{A}}(\mathcal{H})}$.*

Proof. Suppose $\tau_{\mathcal{A}}(u, s) \in obs_{\mathcal{H}}$. Then there exists a state h' of $\mathcal{H}$ such that $h_0 \overset{\tau^I_{\mathcal{A}}(u,s,r_0)/\tau^O_{\mathcal{A}}(u,s,r_0)}{\Rightarrow} h'$. Let r' be the unique state of $\mathcal{A}$ such that $r_0 \overset{u/s}{\rightsquigarrow} r'$. By Claim 2, $(r_0, h_0) \overset{u/s}{\Rightarrow} (r', h')$. Hence $(u, s) \in obs_{\gamma_{\mathcal{A}}(\mathcal{H})}$, as required.

The following key lemma, which builds on the previous lemmas in this subsection, estabishes the duality of the concretization and abstraction operators.

Lemma 4. *Suppose $\alpha_{\mathcal{A}}(\mathcal{M}) \leq \mathcal{H}$. Then $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$.*

Proof. Let $(u, s) \in obs_{\mathcal{M}}$. It suffices to prove $(u, s) \in obs_{\gamma_{\mathcal{A}}(\mathcal{H})}$. By Lemma 2, $\tau_{\mathcal{A}}(u, s) \in obs_{\alpha_{\mathcal{A}}(\mathcal{M})}$. By the assumption, $\tau_{\mathcal{A}}(u, s) \in obs_{\mathcal{H}}$. Hence, by Lemma 3, $(u, s) \in obs_{\gamma_{\mathcal{A}}(\mathcal{H})}$.

The converse of Lemma 4 also holds, but we have not included the proof here, since we do not need this result in the present paper.

3.2 The Behavior of the Mapper Module

We are now prepared to formalize the ideas of Example 2 and establish that, by using an intermediate mapper component, a learner can indeed learn a correct model of the behavior of an implementation. To begin with, we describe how a mapper $\mathcal{A} = \langle I, O, R, r_0, \delta, X, Y, \psi \rangle$ fully determines the behavior of the intermediate mapper component. The mapper component maintains a state variable of type R, which initially is set to r_0. The behavior of the module is defined as follows:

- Whenever the mapper is in a state r and receives an abstract input symbol $x \in X$ from the learner, it nondeterministically picks a concrete input symbol $i \in I$ such that $\psi(r, i) = x$, forwards i as an output query to the implementation, and jumps to state $r' = \delta(r, i)$. If there exists no concrete input i such that $\psi(r, i) = x$, then the mapper returns output $\perp$ to the learner.
- Whenever the mapper is in state r' and receives a concrete answer o from the implementation, it forwards the abstract version $\psi(r', o)$ to the learner and jumps to state $r'' = \delta(r', o)$.
- Whenever the mapper receives a reset query from the learner, it changes its current state to r_0, and forwards a reset query to the implementation.

From the perspective of a learner, an implementation for $\mathcal{M}$ and a mapper for $\mathcal{A}$ together behave exactly like an implementation for $\alpha_{\mathcal{A}}(\mathcal{M})$. Since we have not formalized the notion of behavior for implementation and mapper, the mathematical content of this claim may not be immediately obvious. Clearly, it is routine to describe the behavior of an implementation and a mapper formally as state machines in some concurrency formalism, for instance in Milner's CCS [37]

or another process algebra [10]. More precisely, we may define, for each Mealy machine $\mathcal{M}$, a CCS process $\mathsf{Implementation}(\mathcal{M})$ that describes the behavior of an implementation for $\mathcal{M}$, and for each mapper $\mathcal{A}$ a CCS process $\mathsf{Mapper}(\mathcal{A})$ that models the behavior of a mapper module for $\mathcal{A}$. These two CCS processes may then synchronize via actions taken from $I \cup O \cup \{\mathsf{reset}\}$. If we compose $\mathsf{Implementation}(\mathcal{M})$ and $\mathsf{Mapper}(\mathcal{A})$ using the CCS composition operator $|$, and apply the CCS restriction operator $\setminus$ to internalize all communications between the two processes, the resulting CCS process is weakly bisimilar to the CCS process $\mathsf{Implementation}(\alpha_{\mathcal{A}}(\mathcal{M}))$:

$$(\mathsf{Implementation}(\mathcal{M}) \mid \mathsf{Mapper}(\mathcal{A})) \setminus (I \cup O \cup \{\mathsf{reset}\}) \approx \mathsf{Implementation}(\alpha_{\mathcal{A}}(\mathcal{M})).$$

It is in this precise, formal sense that one should read the following theorem. The reason why we do not refer to the CCS formalization in the statement and proof of this theorem is that we feel that the resulting notational overhead would obscure rather than clarify.

Theorem 1. *An implementation for $\mathcal{M}$ and a mapper for $\mathcal{A}$ together behave like an implementation for $\alpha_{\mathcal{A}}(\mathcal{M})$.*

Proof. Initially, the state of the implementation for $\mathcal{M}$ is q_0 and the current state of the mapper is r_0, which is consistent with the initial state (q_0, r_0) of an implementation for $\alpha_{\mathcal{A}}(\mathcal{M})$.

Suppose that the current state is (q, r) and an output query $x \in X$ arrives. If there exists a concrete input i such that $\psi(r, i) = x$, then the mapper nondeterministically picks one such i, passes it on to the implementation (which accepts concrete input symbols) and jumps to state $r' = \delta(r, i)$. In response, the implementation picks a transition $q \xrightarrow{i/o} q'$, jumps to state q' and returns the concrete output symbol $o \in O$ to the mapper. Next, the mapper computes the corresponding abstract value $\psi(r', o) = y$, forwards y to the learner, and jumps to state $r'' = \delta(r', o)$. By inspection of the first transition rule for $\alpha_{\mathcal{A}}(\mathcal{M})$, it follows that the implementation for $\mathcal{M}$ and mapper together behave like an implementation for $\alpha_{\mathcal{A}}(\mathcal{M})$ in this case. If there exists no concrete input i such that $\psi(r, i) = x$, then the mapper returns output $\perp$ to the learner. By inspection of the second transition rule for $\alpha_{\mathcal{A}}(\mathcal{M})$, it follows that the implementation of $\mathcal{M}$ and mapper together again behave like an implementation for $\alpha_{\mathcal{A}}(\mathcal{M})$.

Now suppose the mapper receives a reset query from the learner. Then the mapper moves to its initial state r_0 and forwards the reset query to the implementation, who also returns to its initial state q_0. This behavior is consistent with the behavior of an implementation for $\alpha_{\mathcal{A}}(\mathcal{M})$, which returns to its initial state (q_0, r_0) upon receiving a reset query.

In order to learn $\alpha_{\mathcal{A}}(\mathcal{M})$, a learner does not only need an implementation of $\alpha_{\mathcal{A}}(\mathcal{M})$ but also an oracle for $\alpha_{\mathcal{A}}(\mathcal{M})$. In practice, LearnLib "approximates" oracles by computing, from the latest hypothesis, long test sequences using standard methods like state cover, transition cover or the W-method, and appying these test sequences on the implementation to check if the produced output

agrees with the output predicted by the model. In our experiments, described in Section 6, the computed test sequences were sent to the mapper, that is, the implementation of $\alpha_{\mathcal{A}}(\mathcal{M})$. Our mapper used randomization to select concrete input symbols for the abstract input symbols contained in LearnLib queries. Although the initial results in our experiments were very positive, more research will be required to find out whether this is a good "approximation" of an oracle for $\alpha_{\mathcal{A}}(\mathcal{M})$. In [2], we explored the theory of an alternative approach, in which the learner sends an hypothesis $\mathcal{H}$ to the mapper, which in turn forwards the concretization $\gamma_{\mathcal{A}}(\mathcal{H})$ to an oracle for $\mathcal{M}$, and returns an abstract version of the response given by the oracle to the learner.

Given that we have constructed both an implementation and an oracle for $\alpha_{\mathcal{A}}(\mathcal{M})$, and assuming that $\alpha_{\mathcal{A}}(\mathcal{M})$ is finite and behavior deterministic, LearnLib should succeed in inferring a deterministic Mealy machine $\mathcal{H}$ that is equivalent to $\alpha_{\mathcal{A}}(\mathcal{M})$. Whenever $\mathcal{H}$ is correct for $\alpha_{\mathcal{A}}(\mathcal{M})$, then it follows by Lemma 4 that $\gamma_{\mathcal{A}}(\mathcal{H})$ is correct for $\mathcal{M}$.

In general, $\gamma_{\mathcal{A}}(\mathcal{H})$ will not be behavior deterministic: it provides an over-approximation of the behavior of $\mathcal{M}$. For this reason we replaced *equivalence queries* by *inclusion queries* in our learning framework.

Example 6. The mapper $\mathcal{A}$ for Mealy machine $\mathcal{M}_{COM}$ introduced in Example 3 is just right: Mealy machine $\alpha_{\mathcal{A}}(\mathcal{M}_{COM})$ is simple and has only two states, it is deterministic, and if we concretize it again then the resulting Mealy machine $\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(\mathcal{M}_{COM}))$ is observation equivalent to the original $\mathcal{M}_{COM}$. However, in some applications it may be difficult to predict exactly which output will occur when, and in such cases it makes sense to define mappers that abstract away information from the SUT. With such a mapper we will not learn a Mealy machine that is observation equivalent to the Mealy machine of the teacher, but rather an over-approximation.

In order to illustrate this, we consider an alternative mapper for $\mathcal{M}_{COM}$:

$$\mathcal{A}' = \langle I, O, \{\bot\} \cup \mathbb{N}, \bot, \delta', X', O, \psi' \rangle.$$

The sets I and O are the same as for $\mathcal{M}_{COM}$. Mapper $\mathcal{A}'$ only records the selected value of the identifier and just ignores the sequence number parameter. The state of $\mathcal{A}'$ only changes when the first $REQ(id, sn)$ input arrives:

$$\delta'(\bot, REQ(id, sn)) = id,$$
$$\delta'(id, REQ(id', sn')) = id.$$

Output actions do not change the state of $\mathcal{A}'$: $\delta'(r, o) = r$, for $r \in \{\bot\} \cup \mathbb{N}$ and $o \in O$. There are two abstract input symbols: $X' = \{REQ(\mathrm{C}), REQ(\mathrm{O})\}$. In the initial state the abstraction function maps to $REQ(\mathrm{C})$, and for subsequent actions it only records whether the identifier is correct:

$$\psi'(\bot, REQ(id, sn)) = REQ(\mathrm{C}),$$
$$\psi'(id, REQ(id', sn')) = REQ(\mathrm{ID}),$$

where ID = **if** $id' = id$ **then** C **else** O. For outputs ψ' acts as the identity function.

The reader can easily check that $\alpha_{\mathcal{A}'}(\mathcal{M}_{COM})$ is behaviorally equivalent to the nondeterministic Mealy machine $\mathcal{H}'_{COM}$ displayed in Figure 4. The con-

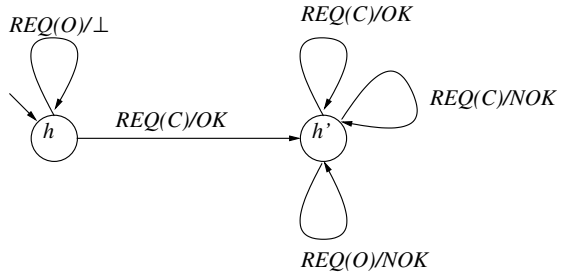

Fig. 4. Minimal Mealy machine $\mathcal{H}'_{COM}$ equivalent to $\alpha_{\mathcal{A}'}(\mathcal{M}_{COM})$

cretization $\gamma_{\mathcal{A}'}(\mathcal{H}'_{COM})$ has the following transitions, for all $id, id', sn \in \mathbb{N}$ with $id' \neq id$ (only transitions reachable from the initial state are listed):

$$
\begin{aligned}
(\bot, h) &\xrightarrow{REQ(id,sn)/OK}{}' (id, h') \\
(id, h') &\xrightarrow{REQ(id,sn)/OK}{}' (id, h') \\
(id, h') &\xrightarrow{REQ(id,sn)/NOK}{}' (id, h') \\
(id, h') &\xrightarrow{REQ(id',sn)/NOK}{}' (id, h')
\end{aligned}
$$

By Lemma 4, we have $\mathcal{M}_{COM} \leq \gamma_{\mathcal{A}'}(\mathcal{H}'_{COM})$. This time $\gamma_{\mathcal{A}'}(\mathcal{H}'_{COM})$ is an over-approximation of $\mathcal{M}_{COM}$ since, for instance, $\gamma_{\mathcal{A}'}(\mathcal{H}'_{COM})$ has a trace $REQ(1,2)/OK$ $REQ(1,2)/OK$, which is not allowed by $\mathcal{M}_{COM}$.

4 Symbolic Abstraction

Even though our general approach for using abstraction in automata learning is phrased most naturally at the semantic level, an actual implementation of our approach requires a syntactic (symbolic) representation of Mealy machines and abstractions. Therefore, in this section, we present a general syntax for symbolic representation of Mealy machines and mappers.

We assume a language with (typed) variables, function, predicate, and constant symbols. We assume that each variable v comes equipped with a type $\mathsf{type}(v)$, which is the (nonempty) set of values that it may take. We postulate that for each variable v there is a primed version v', which has the same type. If V is a set of variables then we write V' to denote the set $\{v' \mid v \in V\}$. We assume that, using the variables, function, predicate, and constant symbols, it is possible to construct terms and formulas. Each term t has an associated type $\mathsf{type}(t)$. We use $\equiv$ to denote syntactic equality of terms. If V is a set of variables, then a *valuation* for V is a function that maps each variable in V to an

element of its domain. We write $\mathsf{Val}(V)$ for the set of all valuations for V. If ξ is a valuation for V and φ is a formula with (free) variables in V, then we write $\xi \models \varphi$ to denote that ξ satisfies φ. Similarly, if t is a term then we write $[\![t]\!]_\xi$ for the value to which t evaluates under valuation ξ. If $V' \subseteq V$ then $\xi\lceil V'$ denotes the restriction of ξ to the variables in V'. If $v_1, \ldots, v_n$ are variables in V and $t_1, \ldots, t_n$ are terms, then we write $\xi[v_1, \ldots, v_n := t_1, \ldots, t_n]$ for the valuation in which all variables have the same values as in ξ except for $v_1, \ldots, v_n$ which are evaluated to $[\![t_1]\!]_\xi, \ldots, [\![t_n]\!]_\xi$, respectively.

We employ a slight variation of Jonsson's [29] approach for specification of distributed systems and define a *symbolic Mealy machine* by means of a program-like notation with guarded multiple assignments. Each assignment statement is labeled with two events which denote reception and transmission of a message.

An event signature specifies the possible interactions between a symbolic Mealy machine and its environment.

Definition 5 (Event signature). *An* event term *is an expression of the form $\varepsilon(p_1, \ldots, p_m)$, where ε is a symbol referred to as the* event primitive*, and $p_1, \ldots p_m$ pairwise different variables referred to as* parameters*. An* event signature *Σ is a pair $\langle T_I, T_O \rangle$, where T_I and T_O are disjoint, finite sets of event terms. We require that the event primitives of different event terms in $T_I \cup T_O$ are distinct.*

Using event signatures, we can define the notion of a symbolic Mealy machine.

Definition 6 (SMM). *A* symbolic Mealy machine (SMM) *is a tuple $\mathcal{M}_S = \langle \Sigma, V, \Theta, \Delta \rangle$, where*

- *$\Sigma = \langle T_I, T_O \rangle$ is an event signature,*
- *V is a finite set of variables, referred to as* state variables*, disjoint from the set of parameters of Σ,*
- *Θ is a formula, the* initial condition*, with (free) variables in V. We require that there is a unique valuation $q_0 \in \mathsf{Val}(V)$ such that $q_0 \models \Theta$, and*
- *Δ is a finite set of* transitions*, each of form*

$$\textbf{event } \varepsilon_I(p_1, \ldots, p_m) \textbf{ when } \varphi \textbf{ event } \varepsilon_O(p_{m+1}, \ldots, p_l)$$

where $\varepsilon_I(p_1, \ldots, p_m) \in T_I$, $\varepsilon_O(p_{m+1}, \ldots, p_l) \in T_O$, $\{p_1, \ldots, p_m\} \cap \{p_{m+1}, \ldots, p_l\} = \emptyset$, and φ is a formula with (free) variables in $\{p_1, \ldots, p_l\} \cup V \cup V'$.

Example 7 (Symbolic representation of $\mathcal{M}_{COM}$). We illustrate how Mealy machine $\mathcal{M}_{COM}$ can be described as a SMM $\mathcal{SM}_{COM}$.

- $\Sigma = \langle \{REQ(\mathsf{p_1}, \mathsf{p_2})\}, \{OK, NOK\} \rangle$, where REQ, OK and NOK are event primitives and $\mathsf{p_1}$ and $\mathsf{p_2}$ are parameters of type $\mathbb{N}$.
- $V = \{\mathsf{ID}, \mathsf{SN}, \mathsf{INIT}\}$, where ID and SN have type $\mathbb{N}$ and INIT has type $\mathbb{B}$. Variable ID stores the current session identifier, SN stores the current sequence number, and INIT records whether a session needs to be initialized.
- Initially, ID and SN are 0 and a session needs to be initialized:

$$\Theta \quad \equiv \quad \mathsf{ID} = 0 \wedge \mathsf{SN} = 0 \wedge \mathsf{INIT}.$$

– Set Δ contains three transitions:

$$\begin{aligned}
&\textbf{event } REQ(\mathsf{p}_1,\mathsf{p}_2) \textbf{ when } \mathsf{INIT} \wedge \mathsf{ID}' = \mathsf{p}_1 \wedge \mathsf{SN}' = \mathsf{p}_2 \wedge \neg\mathsf{INIT}' \\
&\qquad\qquad \textbf{event } OK \\
&\textbf{event } REQ(\mathsf{p}_1,\mathsf{p}_2) \textbf{ when } \neg\mathsf{INIT} \wedge \mathsf{p}_1 = \mathsf{ID} \wedge \mathsf{p}_2 = \mathsf{SN}+1 \wedge \\
&\qquad\qquad \mathsf{ID}' = \mathsf{ID} \wedge \mathsf{SN}' = \mathsf{p}_2 \wedge \neg\mathsf{INIT}' \textbf{ event } OK \\
&\textbf{event } REQ(\mathsf{p}_1,\mathsf{p}_2) \textbf{ when } \neg\mathsf{INIT} \wedge (\mathsf{p}_1 \neq \mathsf{ID} \vee \mathsf{p}_2 \neq \mathsf{SN}+1) \wedge \\
&\qquad\qquad \mathsf{ID}' = \mathsf{ID} \wedge \mathsf{SN}' = \mathsf{SN} \wedge \neg\mathsf{INIT}' \textbf{ event } NOK
\end{aligned}$$

Every transition contains an input event, an output event, and a **when** clause that determines the conditions that need to hold in the current and the next state. For example, in the first transition a $REQ(\mathsf{p}_1,\mathsf{p}_2)$ input triggers an OK output whenever the session needs to be initialized. In the next state, the initialization is completed by assigning to ID the value of p_1 and to SN the value of p_2. □

The semantics of symbolic Mealy machines is defined, in a straightforward manner, in terms of Mealy machines.

Definition 7 (Semantics of SMM). *The semantics of an event term* $\varepsilon(p_1,\ldots,p_m)$ *is the set*

$$[\![\varepsilon(p_1,\ldots,p_m)]\!] \;=\; \{\varepsilon(d_1,\ldots,d_m) \mid d_1 \in \mathsf{type}(p_1),\ldots,d_m \in \mathsf{type}(p_m)\}.$$

The semantics of a set T *of event terms is defined by pointwise extension:*

$$[\![T]\!] \;=\; \bigcup_{\varepsilon(p_1,\ldots,p_m)\in T} [\![\varepsilon(p_1,\ldots,p_m)]\!].$$

Let $\mathcal{M}_S = \langle \Sigma, V, \Theta, \Delta\rangle$ *be a symbolic Mealy machine with* $\Sigma = \langle T_I, T_O\rangle$*. The semantics of* $\mathcal{M}_S$*, notation* $[\![\mathcal{M}_S]\!]$*, is the Mealy machine* $\langle I, O, Q, q_0, \rightarrow\rangle$ *where*

– $I = [\![T_I]\!]$,
– $O = [\![T_O]\!]$,
– $Q = \mathsf{Val}(V)$,
– $q_0 \in \mathsf{Val}(V)$ *is the unique valuation satisfying* $q_0 \models \Theta$*, and*
– $\rightarrow \subseteq Q \times I \times O \times Q$ *is the smallest set that satisfies*

$$\frac{\begin{array}{c}(\textbf{event } \varepsilon_I(p_1,\ldots,p_m) \textbf{ when } \varphi \textbf{ event } \varepsilon_O(p_{m+1},\ldots,p_l)) \in \Delta \\ \forall j \le l, \xi(p_j) = d_j \\ \forall v \in V, \xi(v) = q(v) \text{ and } \xi(v') = q'(v) \\ \xi \models \varphi\end{array}}{q \xrightarrow{\varepsilon_I(d_1,\ldots,d_m)/\varepsilon_O(d_{m+1},\ldots,d_l)} q'}$$

The reader may check that the semantics of the symbolic Mealy machine $\mathcal{SM}_{COM}$ described in Example 7 indeed yields the Mealy machine $\mathcal{M}_{COM}$ of Example 2: the only difference is that states (id, sn, b) of $\mathcal{M}_{COM}$ correspond to valuations in

$\mathcal{SM}_{COM}$, in which variable ID has value id, variable SN has value sn, and variable INIT has value b. In this article, we only consider symbolic Mealy machines whose semantics is input enabled, as required for a Mealy machine.

In the same way as symbolic Mealy machines constitute a syntactic representation of Mealy machines, the definition below introduces *symbolic mappers* as a syntactic representation of mappers. The abstract event signature of a symbolic mapper is the same as its concrete event signature, except that the parameters have a different (typically smaller) domain.

Definition 8 (SM). *Let $\Sigma_c = \langle T_I, T_O \rangle$ be an event signature. A* symbolic mapper (SM) *for Σ_c is a structure $\mathcal{A}_S = \langle \Sigma_c, V, \Theta, \Delta, \Sigma_a, \Psi \rangle$, where*

- *$V = \{v_1, \ldots, v_n\}$ is a finite set of variables, disjoint from the set of parameters of Σ_c,*
- *Θ is a formula, the* initial condition*, whose free variables are in V. We require that there exists a unique valuation $r_0 \in \mathsf{Val}(V)$ such that $r_0 \models \Theta$,*
- *Δ is a finite set of* transitions *given by*

$$\textbf{event } \varepsilon(p_1, \ldots, p_m) \textbf{ when } \varphi \textbf{ do } \langle v_1, \ldots, v_n \rangle := \langle t_1, \ldots, t_n \rangle$$

  where $\varepsilon(p_1, \ldots, p_m) \in T_I \cup T_O$, φ is the guard*, a formula with variables in $V \cup \{p_1, \ldots, p_m\}$, and $t_1, \ldots, t_n$ are terms with variables in $V \cup \{p_1, \ldots, p_m\}$. We require that $\mathcal{A}_S$ is input and output enabled: for each $\varepsilon(p_1, \ldots, p_m) \in T_I \cup T_O$, the disjunction of the set of guards of transitions for that event primitive is equivalent to true. Furthermore, we require that $\mathcal{A}_S$ is deterministic: whenever we have two different transitions for the same event primitive then the conjunction of their guards is equivalent to false,*
- *$\Sigma_a = \langle T_X, T_Y \rangle$ is an event signature, referred to as the* abstract event signature*. We require that, for each $\varepsilon(p_1, \ldots, p_m) \in T_I$, T_X contains an element $\varepsilon(q_1, \ldots, q_m)$. Similarly, we require that, for each $\varepsilon(p_1, \ldots, p_m) \in T_O$, T_Y contains a corresponding element $\varepsilon(q_1, \ldots, q_m)$.*
- *Ψ is a finite set of* event abstractions *which contains, for each event term $\varepsilon(p_1, \ldots, p_m) \in T_I \cup T_O$, an expression $\varepsilon(e_1, \ldots, e_m)$, where, for each j, e_j is a term with variables in $V \cup \{p_1, \ldots, p_m\}$ and $\mathsf{type}(e_j) = \mathsf{type}(q_j)$.*

Example 8 (Symbolic mapper). We illustrate how the mapper $\mathcal{A}$ for Mealy machine $\mathcal{M}_{COM}$, which we defined in Example 3, can also be described as a symbolic mapper $\mathcal{A}_S$:

- $\Sigma_c = \langle \{REQ(\mathsf{p_1}, \mathsf{p_2})\}, \{OK, NOK\} \rangle$, where REQ, OK and NOK are event primitives and $\mathsf{p_1}$ and $\mathsf{p_2}$ are parameters of type $\mathbb{N}$.
  Note that Σ_c equals the event signature Σ of Example 7.
- $V = \{curId, curSn\}$, where $curId$ and $curSn$ are variables of type $\mathbb{N} \cup \{\bot\}$, used to store the identifier and sequence number, respectively, of the current session.
- In the initial state both variables are undefined:

$$\Theta \quad \equiv \quad curId = \bot \wedge curSn = \bot \; .$$

– Set Δ contains five transitions:

event $REQ(\mathsf{p}_1, \mathsf{p}_2)$	**when** $curId = \bot$
	do $\langle curId, curSn\rangle := \langle \mathsf{p}_1, \mathsf{p}_2\rangle$
event $REQ(\mathsf{p}_1, \mathsf{p}_2)$	**when** $curId \neq \bot \wedge \mathsf{p}_1 = curId \wedge \mathsf{p}_2 = curSn + 1$
	do $\langle curSn\rangle := \langle \mathsf{p}_2\rangle$
event $REQ(\mathsf{p}_1, \mathsf{p}_2)$	**when** $curId \neq \bot \wedge (\mathsf{p}_1 \neq curId \vee \mathsf{p}_2 \neq curSn + 1)$
	do $\langle\rangle := \langle\rangle$
event OK	**when** TRUE **do** $\langle\rangle := \langle\rangle$
event NOK	**when** TRUE **do** $\langle\rangle := \langle\rangle$

  The first transition, for instance, states that when receiving a REQ input and variable $curId$ still has its initial value, we need to assign the state variables the values received in the input message.
– $\Sigma_a = \langle\{REQ(\mathsf{q}_1, \mathsf{q}_2)\}, \{OK, NOK\}\rangle$, where REQ, OK and NOK are event primitives and q_1 and q_2 are parameters of type {C, O}.
– Set Ψ contains three event abstractions:
  - $REQ($ `if` $curId = \bot \vee curId = \mathsf{p}_1$ `then` C `else` O , `if` $curSn = \bot \vee curSn + 1 = \mathsf{p}_2$ `then` C `else` O $)$
  - OK
  - NOK □

The semantics of symbolic mappers is defined, again in a straightforward manner, in terms of mappers.

Definition 9 (Semantics of SM). *Let $\mathcal{A}_S = \langle \Sigma_c, V, \Theta, \Delta, \Sigma_a, \Psi\rangle$ be a symbolic mapper for Σ_c. Let $\Sigma_a = \langle T_X, T_Y\rangle$. The semantics of $\mathcal{A}_S$, notation $[\![\mathcal{A}_S]\!]$, is the mapper $\mathcal{A} = \langle I, O, R, r_0, \delta, X, Y, \psi\rangle$, where*

– $I = [\![T_I]\!]$,
– $O = [\![T_O]\!]$,
– $R = \mathsf{Val}(V)$,
– *r_0 is the unique valuation satisfying $r_0 \models \Theta$,*
– *δ is given by the rule*

$$\frac{\begin{array}{c}(\textbf{event } \varepsilon(p_1, \ldots, p_m) \textbf{ when } \varphi \textbf{ do } \langle v_1, \ldots, v_n\rangle := \langle t_1, \ldots, t_n\rangle) \in \Delta \\ \forall j \leq m, \xi(p_j) = d_j \qquad r \cup \xi \models \varphi \\ r' = ((r \cup \xi)[v_1, \ldots, v_n := t_1, \ldots, t_n])\lceil V\end{array}}{r \xrightarrow{\varepsilon(d_1, \ldots, d_m)} r'}$$

– $X = [\![T_X]\!]$,
– $Y = [\![T_Y]\!]$, *and*
– *for all $r \in R$, $\varepsilon(p_1, \ldots, p_m) \in T_I \cup T_O$, ξ a valuation of $\{p_1, \ldots, p_m\}$ such that, for $1 \leq j \leq m$, $\xi(p_j) = d_j$, and $\varepsilon(e_1, \ldots, e_m) \in \Psi$,*

$$\psi(r, \varepsilon(d_1, \ldots, d_m)) = \varepsilon([\![e_1]\!]_{r\cup\xi}, \ldots, [\![e_m]\!]_{r\cup\xi}).$$

Given a symbolic mapper $\mathcal{A}_S$ and a Mealy machine (hypothesis) $\mathcal{H}$, we may construct a symbolic Mealy machine $\gamma^S_{\mathcal{A}_S}(\mathcal{H})$ such that $[\![\gamma^S_{\mathcal{A}_S}(\mathcal{H})]\!]$ is isomorphic to $\gamma_{[\![\mathcal{A}_S]\!]}(\mathcal{H})$. Since the construction is routine and not required for the remainder of this paper, we leave it to the reader to work out the details.

5 Systematic Construction of Abstractions

The construction of a suitable mapper component is an important part of our technique for generating a model of an SMM $\mathcal{M}_S$. In general, the construction of the mapper will rely on insights into what aspects of the data parameters are important for the behavior of $\mathcal{M}_S$. But it is also possible to present guidelines for constructing them systematically, from which also automated support can be developed. In this section, we suggest a set of such guidelines.

To simplify our presentation, we assume that output event primitives do not have parameters, as is the case, e.g., in Example 8. Then the main purpose of the mapper is to provide an abstraction of the parameters of input symbols, which preserves the information that determines which output symbols will subsequently be generated in an observation. More precisely if (u, s) and (u', s') are different observations of $\mathcal{M}_S$, which the mapper abstracts to $\tau_{\mathcal{A}}(u, s) = (U, S)$ and $\tau_{\mathcal{A}}(u', s') = (U', S')$, then $S \neq S'$ should imply $U \neq U'$, otherwise the abstraction $\alpha_{\mathcal{A}}(\mathcal{M}_S)$ will behave nondeterministically, something that the learning algorithm is not designed for. The requirement to produce a behavior deterministic abstraction suggests a methodology for constructing mappers, in which observed nondeterminism in $\alpha_{\mathcal{A}}(\mathcal{M}_S)$ triggers a modification of the mapper.

One can start from an initial mapper, whose event abstractions are trivial, i.e., they map any value of any parameter in any input symbol to a single abstract value. Whenever a sequence of output queries shows that the composition of mapper and $\mathcal{M}_S$ is nondeterministic, i.e., there is a pair of observations, (u, s) and (u', s'), such that with $\tau_{\mathcal{A}}(u, s) = (U, S)$ and $\tau_{\mathcal{A}}(u', s') = (U', S')$ we have $S \neq S'$ but $U = U'$, then some event abstraction that contributes to generating U or U' must be refined. This refinement is constructed by first performing additional output queries to determine in what way the parameters in u and u' cause S and S' to be different. In many cases, it is possible to find a particular condition that determines whether the output will be S or S'. This condition is then introduced into the mapper in order to differentiate between $\tau_{\mathcal{A}}(u, s)$ and $\tau_{\mathcal{A}}(u', s')$. In the case that the new condition refers to parameters in different symbols of u and u', variables must be introduced into the mapper that remember received data values, in order that the new condition can refer to them.

Let us illustrate how these guidelines can be applied in Example 8. We start from an initial (too coarse) abstraction, in which the mapper does not distinguish between different parameter values in input symbols of form $REQ(d_1, d_2)$. By performing output queries, we discover that the resulting composition of mapper and $\mathcal{M}_S$ is nondeterministic. Namely, an input of form $REQ(d_1, d_2)$ may give rise either to an output OK or an output NOK. Additional investigation by means of output queries, in order to find a distinction between these two cases, reveals that the OK output occurs precisely in the case that

- d_1 occurred in the first input of form $REQ(d_1', d_2')$ (with $d_1' = d_1$), and
- $d_2 - 1$ occurred in the most recent input of form $REQ(d_1', d_2')$, which resulted in an OK response from $\mathcal{M}_S$.

As a result of these insights, we let the mapper have

– one variable (say, *curId*) which stores the value of d'_1 in the first input of form $REQ(d'_1, d'_2)$, and
– one variable (say, *curSn*) which stores the value of d'_2 whenever an input of form $REQ(d'_1, d'_2)$ arrives and results in an OK response.

Furthermore, we refine the event abstraction for $REQ(\mathsf{p}_1, \mathsf{p}_2)$, as follows.

– p_1 is mapped to one abstract value (say, C) if its value is equal to the value of *curId*, and to another value (say, O) otherwise.
– p_2 is mapped to one abstract value (say, C) if its value is equal to the value of $curSn + 1$, and to another value (say, O) otherwise.

By completing the mapper based on this abstraction, e.g., also investigating how to handle initialization of variables, we obtain the mapper that is presented in Example 8.

In [1], we show how, following the approach sketched above, mappers can be constructed fully automatically for a restricted class of symbolic Mealy machines in which one can test for equality of data parameters, but no operations on data are allowed.

6 Experiments

We have implemented and applied our approach to infer models of two implemented standard protocols: the Session Initiation Protocol (SIP) and the Transmission Control Protocol (TCP). In order to have access to a large number of standard communication protocols, for evaluation of inference techniques, we used the protocol simulator ns-2 [39], which provides implementations of many protocols, to serve as System Under Test (SUT). Messages are represented as C++ structures, saving us the trouble of parsing messages represented as bit-strings. As learner, we used an efficient implementation of the L^* algorithm in LearnLib [42, 36], a tool developed at the Technical University of Dortmund. LearnLib also provides several implementations of model-based test algorithms in order to realize equivalence queries, including random test suites of user-controlled size. Hence, in our experiments, the teacher consists of an SUT, which is a protocol implemented in ns-2, in combination with a model-based test algorithm implemented in LearnLib. We postulate that the behavior of the SUT can be modelled as a Mealy machine (cf. the notion of *test hypothesis* from model-based testing [48]) and our task is to learn this unknown Mealy machine.

6.1 The Session Initiation Protocol (SIP)

SIP is an application layer protocol for creating and managing multimedia communication sessions [44]. Although a lot of documentation is available, there is no reference model in the form of a state machine. We aimed to infer the behavior of a SIP Server entity when setting up connections with a SIP Client. We represented the messages that can be sent between a SIP Client and a SIP Server by

means of the event signature $\Sigma_{SIP} = \langle T_I, T_O \rangle$. Set T_I contains event terms of the form $Method(CallId, CSeq, Via)$, where $Method = \{\text{INVITE}, \text{PRACK}, \text{ACK}\}$ is the set of input event primitives, which correspond to the different types of requests that can be made by the client:

- an INVITE request is an initial request needed for the session establishment. It indicates that a SIP Client wants to establish a connection with the SIP Server. This activity can be compared with dialing someone's telephone number.
- a PRACK request is an acknowledgement, which is used to confirm provisional responses that could have been lost otherwise.
- an ACK request confirms that a Client has received a final response to an INVITE request. Unlike PRACK, an ACK request does not have a response.

Set T_O contains event terms of the form $StatusCode(CallId, CSeq, Via)$, where $StatusCode = \{100, 180, 183, 200, 481, 486\}$ is the set of output event primitives. The three digit status codes indicate the outcome generated by the Server in response to a previous request by the Client:

- *1xx* responses are provisional responses. A provisional response is sent when the associated request was received but the request still needs to be processed. Possible *1xx* responses are *100* (Trying), *180* (Ringing), which means that the recipient's phone is ringing, and *183* (Session Progress),
- *2xx* responses are positive final responses. They indicate that the request was successful. A *200* (OK) response is sent when a user accepts invitation to a session, and
- *4xx* responses are negative final responses. They indicate that the request contains bad syntax or cannot be fulfilled at the Server. Possible *4xx* responses are *481* (Call/Transaction Does Not Exist) and *486* (Busy Here).

A typical interaction between a Client and a Server is visualized in Table 1.

Client	**Server**
INVITE(*CallId*:4, *CSeq*:1, *Via*:1.1.2;branch=z9hG4bK3) →	
	← 100(*CallId*:4, *CSeq*:1, *Via*:1.1.2;branch=z9hG4bK3) Trying
	← 183(*CallId*:4, *CSeq*:1, *Via*:1.1.2;branch=z9hG4bK3) Session Progress
PRACK(*CallId*:4, *CSeq*:2, *Via*:1.1.2;branch=z9hG4bK3) →	
	← 200(*CallId*:4, *CSeq*:2, *Via*:1.1.2;branch=z9hG4bK3) OK
ACK(*CallId*:4, *CSeq*:1, *Via*:1.1.2;branch=z9hG4bK3) →	

Table 1. Typical session establishment in SIP

All of the above event terms have the same parameters:

– *CallId* is a unique session identifier,
– *CSeq* is a sequence number that orders transactions in a session, and
– *Via* specifies the transport path that is used for the transaction. The *Via* parameter is a pair, consisting of a default address and a variable branch.

The actual messages that are used within SIP carry some additional parameters, specifying the addresses of the originator and receiver of a request, and the address where the Client wants to receive input messages. These parameters must be pre-configured in a session with ns-2, so they are set to constant values throughout the experiment, and play no role in the learning. The parameters *Via*, *CallId*, and *CSeq* are potentially interesting parameters. A priori, they can be handled as parameters from a large domain, on which test for equality and potentially incrementation can be performed.

The SUT does not always respond to each input message, and sometimes responds with more than one message. To stay within the Mealy machine formalism, set T_I contains an additional event term NIL(), which denotes the absence of input (in order to allow sequences of outputs), and set T_O contains an additional event term TIMEOUT(), denoting the absence of output.

Following Definition 8, a symbolic mapper $\mathcal{A}_S$ for SIP can be defined as follows. Monitoring of output queries, as described in Section 5 reveals that for each of these parameters, the ns-2 SIP implementation remembers the value which is received in the first INVITE message (presumably, it is interpreted as parameters of the connection that is being established), and also the value received in the most recent input message when producing the corresponding reply. We therefore equip the mapper with six state variables. Variable *firstInviteId* stores the *CallId* parameter of the first *Invite* message, and variable *lastId* stores the *CallId* parameter value of the most recently received message. Variables *firstInviteCSeq* and *lastCSeq* store the analogous values for the *CSeq* parameter, and the variables *firstInviteVia* and *lastVia* for the *Via* parameter. Initially, all six variables have the undefined value $\perp$. Note that we have to remember these six state variables, because all of them are employed to construct the correct reply, i.e., they are needed to map a concrete output message to an abstract output message.

The transitions define when which state variables have to be updated, e.g.

$$\textbf{event}\ \text{INVITE}(CallId, CSeq, Via)\ \textbf{when}\ \mathit{firstInviteId} = \perp$$

$$\textbf{do}\ \langle \mathit{firstInviteId}, \mathit{firstInviteCSeq}, \mathit{firstInviteVia}\rangle := \langle CallId, CSeq, Via\rangle;$$

$$\langle \mathit{lastId}, \mathit{lastCSeq}, \mathit{lastVia}\rangle := \langle CallId, CSeq, Via\rangle$$

states that when receiving an INVITE input and the *firstInviteId* state variables still has its initial value, the mapper needs to assign the *firstInvite* and *last* state variables the values received in the input message. If the *firstInviteId* state variable does not have its initial value when an INVITE input is received, the following transition occurs:

$$\textbf{event}\ \text{INVITE}(CallId, CSeq, Via)\ \textbf{when}\ \mathit{firstInviteId} \neq \perp$$

$$\textbf{do } \langle lastId, lastCSeq, lastVia\rangle := \langle CallId, CSeq, Via\rangle$$

For PRACK and ACK inputs, the update of state variables is defined by

$$\textbf{event } \textit{(PR)ACK}(CallId, CSeq, Via) \textbf{ when } \mathsf{TRUE}$$

$$\textbf{do } \langle lastId, lastCSeq, lastVia\rangle := \langle CallId, CSeq, Via\rangle$$

For event term NIL() and all output event terms no state variables are updated, and we have trivial transitions of the form **when** TRUE **do** $\langle\rangle := \langle\rangle$.

Additional monitoring of output queries reveals that the mapper needs to consider two cases in the abstraction of the *CallId* parameter in input messages: 1) The concrete value of *CallId* is a fresh value or equal to the *firstInviteId* state variable. In this case *CallId* should be mapped to the abstract value FIRST. 2) The concrete value of *CallId* is NOT a fresh value and NOT equal to the *firstInviteId* state variable. In this case *CallId* should be mapped to the abstract value LAST. Both events require the use of the *firstInviteId* state variable. We define the relation between concrete and abstract input symbols by the event abstractions

$$Method(\begin{array}{c}\texttt{if } (\textit{firstInviteId} = \bot \vee \textit{firstInviteId} = CallId)\\ \texttt{then FIRST else LAST}\end{array}, \text{ANY}, \text{ANY}),$$

where $Method$ can be any input event primitive. The input parameters *Via* and *Cseq* are always mapped to the abstract value ANY, since we found that ns-2 always behaves in the same way - no matter which concrete value has been selected. To cope with unexpected values that might be returned by the SUT, different from the values recorded in the state variables, we added an abstract value OTHER. We define the relation between concrete and abstract output symbols by the event abstraction $StatusCode(e_1, e_2, e_3)$, where $StatusCode$ can be any output event primitive,

$$\begin{aligned}
e_1 &= \texttt{if } CallId = \textit{firstInviteId} \texttt{ then FIRST}\\
&\qquad \texttt{elseif } CallId = \textit{lastId} \texttt{ then LAST else OTHER},\\
e_2 &= \texttt{if } CSeq = \textit{firstInviteCSeq} \texttt{ then FIRST}\\
&\qquad \texttt{elseif } CSeq = \textit{lastCSeq} \texttt{ then LAST else OTHER, and}\\
e_3 &= \texttt{if } Via = \textit{firstInviteVia} \texttt{ then FIRST}\\
&\qquad \texttt{elseif } Via = \textit{lastVia} \texttt{ then LAST else OTHER}.
\end{aligned}$$

Since event terms NIL() and TIMEOUT() carry no parameters, the event abstraction for these terms is trivial.

Results. The inference performed by LearnLib needed about one thousand output queries and one equivalence query, and resulted in an abstract model $\mathcal{H}$ with 9 locations and 63 transitions. This model can be found in Appendix B. For presentation purposes, we have also included a simplified version of model $\mathcal{H}$ in Appendix A. In this pruned model, we removed transitions with output

symbol $\perp$ and transitions with an empty input and output symbol, i.e., *NIL/-TIMEOUT*. In Appendix A, we show the pruned abstract model with 9 locations and 48 transitions. For readability, some transitions with same source location, output symbol and next location (but with different input symbols) are merged: the original input method types are listed, separated by a bar (|). Due to space limitations, we have suppressed the (abstract) parameter values. However, the *CallId* parameter of the input messages with abstract value FIRST is depicted in the model with solid transition lines, the remaining transitions have a dashed line pattern. We suppressed all other parameters in the figure.

The abstract model $\mathcal{H}$ does not contain any output parameter value OTHER: apparently all concrete output values generated by the SUT are mapped to either FIRST or LAST. This means that the symbolic Mealy machine $\gamma^S_{\mathcal{A}_S}(\mathcal{H})$ is behavior deterministic, since the abstract output values FIRST and LAST are always mapped to a single concrete value. If $\mathcal{M}$ is a Mealy machine that models the SUT then, according to Lemma 1, $\mathcal{M} \approx \gamma^S_{\mathcal{A}_S}(\mathcal{H})$. Thus, using our approach, we have succeeeded to learn a model that is observation equivalent to the (unknown) model $\mathcal{M}$ of the ns-2 implementation of the SIP protocol.

6.2 The Transmission Control Protocol (TCP)

As a second case study, we have studied the ns-2 implementation of TCP [41, 47]. TCP is a transport layer protocol, which provides reliable and ordered delivery of a byte stream from one computer application to another. It is one of the core protocols of the Internet Protocol Suite. We consider the connection establishment and termination between a Client and a Server, but leave out the data transfer phase. As *SUT*, we consider the ns-2 implementation of the Server component of the protocol. We consider *Request* messages, which are input to the *SUT*, and *Response* messages, which are output by the *SUT*. We ignore in our presentation a number of fields in TCP messages (these are kept to a constant value in our learning experiments) and consider messages of the form *Request/Response*(*Flag*, *SeqNr*, *AckNr*). Parameter *Flag* consists of three bits *SYN*, *ACK*, and *FIN* that define what type of message is sent: *SYN* synchronizes sequences numbers, *ACK* acknowledges the previous *SeqNr*, and *FIN* signals the end of the data transfer phase. Table 2 lists the four possible values for parameter *Flag*.[3] Parameter *SeqNr* is a number that needs to be synchronized with both sides of the connection, and parameter *AckNr* acknowledges a previous sequence number. In addition, there is an input symbol *nil*, which corresponds to the scenario in which the Client does not send a request to the Server, and an output symbol *timeout*, which corresponds to the scenario in which the Server does not generate any response.

To define the mapper, we use information obtained from the standard [41]. The parameters *SeqNr* and *AckNr* are used as sequence numbers, and incre-

[3] Uijen [49] also describes a more general learning experiment in which all possible combinations of the three control bits are allowed, including the so-called Kamikaze packet [41] in which all the flag bits are turned on.

Flag	*SYN*	*ACK*	*FIN*
SYN	1	0	0
SYN+ACK	1	1	0
ACK	0	1	0
ACK+FIN	0	1	1

Table 2. Possible values for *Flag* parameter.

mented with each transmission round in a session. The mapper has four integer state variables, which initially all have value 0. The first two variables, *last_SeqNr_sent* and *last_AckNr_sent*, record the last values of *SeqNr* and *AckNr*, respectively, that have been transmitted to ns-2 in a valid *Request* message. A message is called *valid* if it follows the protocol and increments parameters *SeqNr* and *AckNr* appropriately:

event *Request(Flag, SeqNr, AckNr)*

when $SeqNr = last\_AckNr\_recd \wedge AckNr = last\_SeqNr\_recd + 1$

do $\langle last\_SeqNr\_sent, last\_AckNr\_sent \rangle := \langle SeqNr, AckNr \rangle$

The other two variables, *last_SeqNr_recd* and *last_AckNr_recd*, record the last values of *SeqNr* and *AckNr*, respectively, that have been received from ns-2 in a valid *Response* message:

event *Response(Flag, SeqNr, AckNr)*

when $SeqNr = 0 \vee SeqNr = last\_AckNr\_sent \wedge AckNr = last\_SeqNr\_recd + 1$

do $\langle last\_SeqNr\_recd, last\_AckNr\_recd \rangle := \langle SeqNr, AckNr \rangle$

The occurrence of a *nil* input or a *timeout* output does not change the state of the mapper:

event *nil*() **when** TRUE **do** $\langle\rangle := \langle\rangle$

event *timeout*() **when** TRUE **do** $\langle\rangle := \langle\rangle$

The relation between concrete and abstract symbols is specified by four event abstractions: *nil*(), *timeout*(), *Request*$(Flag, e_1, e_2)$, where

$e_1 =$ `if` $SeqNr = last\_AckNr\_recd$ `then` VALID `else` INVALID,

$e_2 =$ `if` $AckNr = last\_SeqNr\_recd + 1$ `then` VALID `else` INVALID,

and *Response*$(Flag, e_3, e_4)$, where

$e_3 =$ `if` $SeqNr = 0 \vee SeqNr = last\_AckNr\_sent$ `then` VALID `else` INVALID,

$e_4 =$ `if` $AckNr = last\_SeqNr\_sent + 1$ `then` VALID `else` INVALID.

The main information is that in VALID messages, each *AckNr* increments the previously sent *SeqNr* at both sides, and that each *SeqNr* should be that of the previously received *AckNr*. Note that the abstraction function preserves the value of the *Flag* parameter.

Model of Appendix C	Protocol standard
0	LISTEN
1	SYN_RCVD
4	ESTABLISHED
5	CLOSE_WAIT
8	LAST_ACK & CLOSED

Table 3. Correspondence between states in learned model and in protocol standard.

Results. After inference, LearnLib produced a model with 11 locations and $187 = 11 \times 17$ transitions. In order to display the model in this paper, we suppressed transitions with input symbols that have INVALID abstract parameter values and removed transitions labeled with *nil/timeout*. This results in 10 locations and 41 transitions, shown in Appendix C. The model is displayed in a shorthand symbolic representation for readability reasons.

Validation. To validate the learned TCP model, we compared it to the state diagram given in the standard and a well-known textbook [41, 47]. Our setup is restrictive, in that we do not explicitly use triggers, like CONNECT, SEND, LISTEN, CLOSE and that we do not model a RST message. Furthermore, our learned model reflects only setup and closing of connections which are initiated by the Client communicating with the Server. To include setup and closing initiated by the learned Server, we should also have included the above triggers in the set of input symbols of output queries. We therefore compare the learned model of Appendix C with the paths in the state diagram of [41, 47] that correspond to behavior triggered by the Client. Table 3 indicates the resulting correspondence between the states. We observed the following differences and similarities:

- Connection establishment is consistent in the two models.
- The reference model responds to FIN messages when closing a connection, but the learned model does not respond at all to FIN messages, only to FIN+ACK. This reflects a choice made by the implementors of the module in ns-2. It is our impression that this is a common practice in TCP implementations.
- It follows from Table 3 that for several states in the learned model (3, 6, 7, 9, 10) there are no corresponding states in the state diagrams of [41, 47].

7 Conclusions and Future Work

We have presented an approach to infer models of entities in communication protocols, which also handles message parameters. The approach adapts abstraction, as used in formal verification, to the black-box inference setting. This necessitates to define an abstraction together with the local state needed to define it. This makes finding suitable abstractions more challenging, but we have presented techniques for systematically deriving abstractions under restrictions

on what operations the component may perform on data. We have shown the applicability of our approach for inference of (fragments of) realistic communication protocols, by feasibility studies on SIP and TCP, as implemented in the protocol simulator ns-2. Elsewhere, we describe the successful application of our approach for learning models of the Biometric passport [5] and the Bounded Retransmission Protocol [4]. In future work, we intend to apply our approach to larger fragments of SIP and TCP, and also to other protocols. Our work shows how regular inference can infer the influence of data parameters on control flow, and how data parameters are produced. Thus, models generated using our extension are more useful for thorough model-based test generation, than are finite-state models where data aspects are suppressed. In future work, we plan to supply a library of different inference techniques specialized towards different data domains that are commonly used in communication protocols. Initial steps in this direction are already reported in [1, 13, 35].

Acknowledgement We are grateful to Falk Howar from TU Dortmund for his generous LearnLib support, and to Falk Howar and Bernhard Steffen for fruitful discussions.

References

1. F. Aarts, F. Heidarian, H. Kuppens, P. Olsen, and F.W. Vaandrager. Automata learning through counterexample-guided abstraction refinement. In D. Giannakopoulou and D. Méry, editors, *18th International Symposium on Formal Methods (FM 2012), Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 10–27. Springer, August 2012.
2. F. Aarts, F. Heidarian, and F.W. Vaandrager. A theory of abstractions for learning interface automata. In M. Koutny and I. Ulidowski, editors, *23rd International Conference on Concurrency Theory (CONCUR), Newcastle upon Tyne, UK, September 3-8, 2012. Proceedings*, volume 7454 of *Lecture Notes in Computer Science*, pages 240–255. Springer, September 2012.
3. F. Aarts, B. Jonsson, and J. Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In A. Petrenko, J.C. Maldonado, and A. Simao, editors, *22nd IFIP International Conference on Testing Software and Systems, Natal, Brazil, November 8-10, Proceedings*, volume 6435 of *Lecture Notes in Computer Science*, pages 188–204. Springer, 2010.
4. F. Aarts, H. Kuppens, G.J. Tretmans, F.W. Vaandrager, and S. Verwer. Learning and testing the bounded retransmission protocol. In J. Heinz, C. de la Higuera, and T. Oates, editors, *Proceedings 11th International Conference on Grammatical Inference (ICGI 2012), September 5-8, 2012. University of Maryland, College Park, USA*, volume 21 of *JMLR Workshop and Conference Proceedings*, pages 4–18, 2012.
5. F. Aarts, J. Schmaltz, and F.W. Vaandrager. Inference and abstraction of the biometric passport. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation - 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part I*, volume 6415 of *Lecture Notes in Computer Science*, pages 673–686. Springer, 2010.

6. G. Ammons, R. Bodik, and J. Larus. Mining specifications. In *Proc.* 29^{th} *ACM Symp. on Principles of Programming Languages*, pages 4–16, 2002.
7. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.
8. T. Ball and S.K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc.* 29^{th} *ACM Symp. on Principles of Programming Languages*, pages 1–3, 2002.
9. T. Berg, B. Jonsson, and H. Raffelt. Regular inference for state machines with parameters. In Luciano Baresi and Reiko Heckel, editors, *FASE*, volume 3922 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2006.
10. J.A. Bergstra, A. Ponse, and S.A. Smolka, editors. *Handbook of Process Algebra*. North-Holland, 2001.
11. M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*. Springer Verlag, 2004.
12. Y. Brun and M.D. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE'04: 26th Int. Conf. on Software Enginering*, May 2004.
13. S. Cassel, F. Howar, B. Jonsson, M. Merten, and B. Steffen. A succinct canonical register automaton model. In Tevfik Bultan and Pao-Ann Hsiung, editors, *Automated Technology for Verification and Analysis, 9th International Symposium, ATVA 2011, Taipei, Taiwan, October 11-14, 2011. Proceedings*, volume 6996 of *Lecture Notes in Computer Science*, pages 366–380. Springer, 2011.
14. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.
15. J.M. Cobleigh, D. Giannakopoulou, and C.S. Pasareanu. Learning assumptions for compositional verification. In *Proc. TACAS '03,* 9^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer Verlag, 2003.
16. E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.
17. W. Grieskamp, N. Kicillof, K. Stobie, and V. Braberman. Model-Based Quality Assurance of Protocol Documentation: Tools and Methodology. *Software Testing, Verification and Reliability*, 21(1):55–71, 2011.
18. O. Grinchtein. *Learning of Timed Systems*. PhD thesis, Dept. of IT, Uppsala University, Sweden, 2008.
19. O. Grinchtein, B. Jonsson, and M. Leucker. Learning of event-recording automata. In *Proceedings of the Joint Conferences FORMATS and FTRTFT*, volume 3253 of *LNCS*, pages 379–396, September 2004.
20. A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In J.-P. Katoen and P. Stevens, editors, *Proc. TACAS '02,* 8^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 357–370. Springer Verlag, 2002.
21. R. Groz, K. Li, A. Petrenko, and M. Shahbaz. Modular system verification by inference, testing and reachability analysis. In *TestCom/FATES*, volume 5047 of *Lecture Notes in Computer Science*, pages 216–233, 2008.
22. O. Grumberg and H. Veith, editors. *25 Years of Model Checking: History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*. Springer, 2008.

23. A. Hagerer, H. Hungar, O. Niese, and B. Steffen. Model generation by moderated regular extrapolation. In R.-D. Kutsche and H. Weber, editors, *Proc. FASE '02, 5^{th} Int. Conf. on Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 80–95. Springer Verlag, 2002.
24. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. 29^{th} ACM Symp. on Principles of Programming Languages*, pages 58–70, 2002.
25. F. Howar, M. Isberner, B. Steffen, O. Bauer, and B. Jonsson. Inferring semantic interfaces of data structures. In *ISoLA (1): Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I*, volume 7609 of *Lecture Notes in Computer Science*, pages 554–571. Springer, 2012.
26. F. Howar, B. Steffen, and M. Merten. Automata learning with automated alphabet abstraction refinement. In *VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 2011.
27. A. Huima. Implementing conformiq qtronic. In A. Petrenko, M. Veanes, J. Tretmans, and W. Grieskamp, editors, *Proc. TestCom/FATES, Tallinn, Estonia, June, 2007*, volume 4581 of *Lecture Notes in Computer Science*, pages 1–12, 2007.
28. H. Hungar, O. Niese, and B. Steffen. Domain-specific optimization in automata learning. In *Proc. 15^{th} Int. Conf. on Computer Aided Verification*, 2003.
29. B. Jonsson. Compositional specification and verification of distributed systems. *ACM Trans. on Programming Languages and Systems*, 16(2):259–303, 1994.
30. M.J. Kearns and U.V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.
31. K. Li, R. Groz, and M. Shahbaz. Integration testing of distributed components based on learning parameterized I/O models. In E. Najm, J.-F. Pradat-Peyre, and V. Donzeau-Gouge, editors, *FORTE*, volume 4229 of *Lecture Notes in Computer Science*, pages 436–450, 2006.
32. C. Loiseaux, S. Graf, J. Sifakis, A. Boujjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995.
33. D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *Proc. ICSE'08: 30th Int. Conf. on Software Enginering*, pages 501–510, 2008.
34. L. Mariani and M. Pezz. Dynamic detection of COTS components incompatibility. *IEEE Software*, 24(5):76–85, September/October 2007.
35. M. Merten, F. Howar, B. Steffen, S. Cassel, and B. Jonsson. Demonstrating learning of register automata. In C. Flanagan and B. König, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7214 of *Lecture Notes in Computer Science*, pages 466–471. Springer, 2012.
36. M. Merten, B. Steffen, F. Howar, and T. Margaria. Next generation LearnLib. In P.A. Abdulla and K.R.M. Leino, editors, *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 220–223. Springer, 2011.
37. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
38. O. Niese. An integrated approach to testing complex systems. Technical report, Dortmund University, 2003. Doctoral thesis.
39. The Network Simulator NS-2. `http://www.isi.edu/nsnam/ns/`.

40. D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. In J. Wu, S. T. Chanson, and Q. Gao, editors, *Formal Methods for Protocol Engineering and Distributed Systems, FORTE/PSTV*, pages 225–240, Beijing, China, 1999. Kluwer.
41. J. Postel (editor). Transmission Control Protocol - DARPA Internet Program Protocol Specification (RFC 3261), September 1981. Available via `http://www.ietf.org/rfc/rfc793.txt`.
42. H. Raffelt, B. Steffen, T. Berg, and T. Margaria. LearnLib: a framework for extrapolating behavioral models. *STTT*, 11(5):393–407, 2009.
43. R.L. Rivest and R.E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103:299–347, 1993.
44. J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol (RFC 3261), June 2002. Available via `http://www.ietf.org/rfc/rfc3261.txt`.
45. M. Shahbaz, K. Li, and R. Groz. Learning and integration of parameterized components through testing. In A. Petrenko, M. Veanes, J. Tretmans, and W. Grieskamp, editors, *TestCom/FATES*, volume 4581 of *Lecture Notes in Computer Science*, pages 319–334. Springer, 2007.
46. G. Shu and D. Lee. Testing security properties of protocol implementations - a machine learning based approach. In *Proc. ICDCS'07, 27th IEEE Int. Conf. on Distributed Computing Systems, Toronto, Ontario*. IEEE Computer Society, 2007.
47. W.R. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison Wesley Longman, Inc., 1994.
48. J. Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, December 1992.
49. J. Uijen. *Learning Models of Communication Protocols using Abstraction Techniques*. Master thesis, Radboud University Nijmegen and Uppsala University, November 2009.
50. M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*, pages 39–76. Springer, 2008.

A Pruned SIP Model

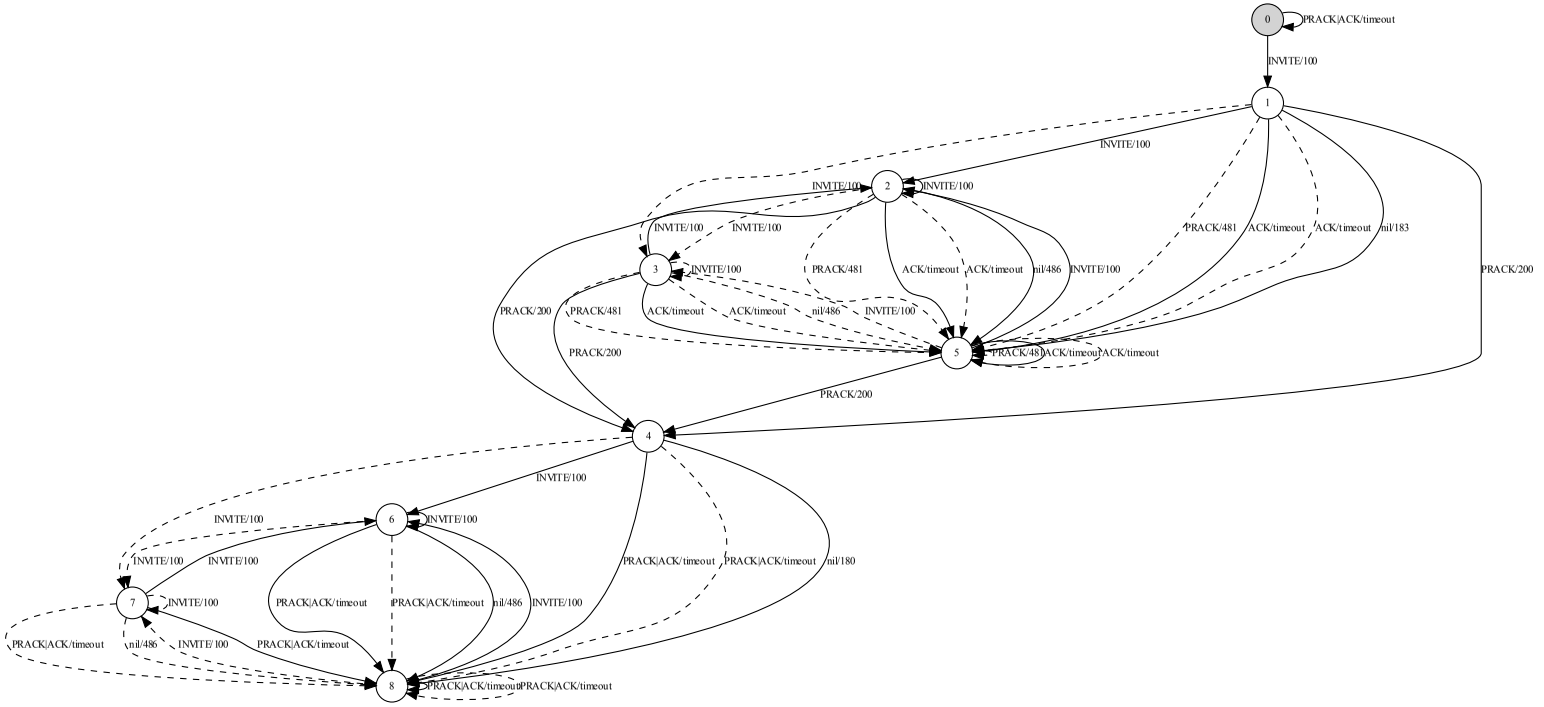

Fig. 5. Pruned SIP model

B Complete SIP Model

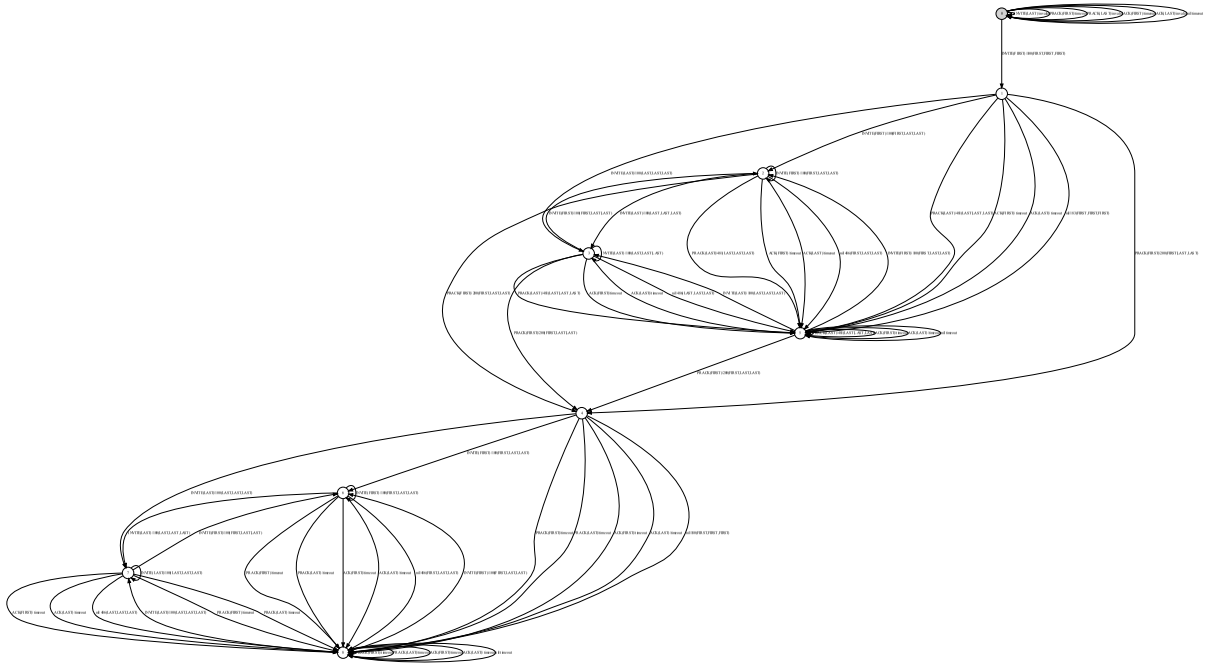

Fig. 6. Complete SIP model

C Model of TCP Server

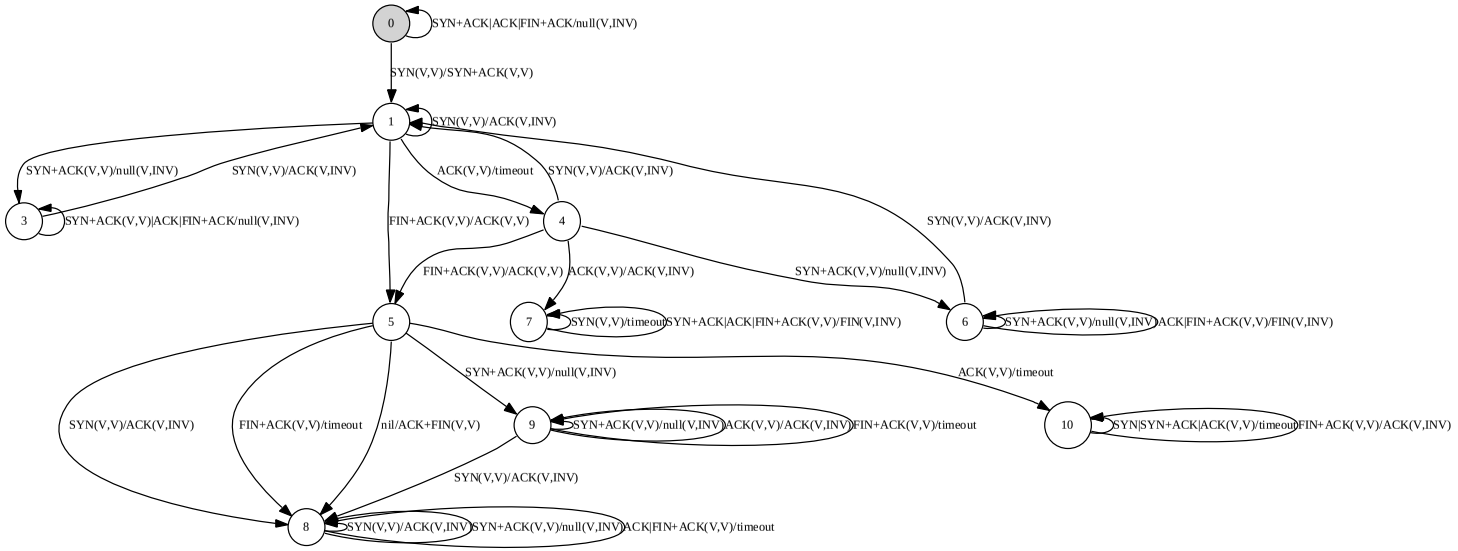

Fig. 7. Learned model of the TCP server. For readability, we write $Flag(SeqNr, AckNr)/Flag'(SeqNr', AckNr')$ instead of $Request(Flag, SeqNr, AckNr)/Response(Flag', SegNr', AckNr')$. Moreover, V represents the VALID equivalence class and INV represents the INVALID equivalence class.

D List of Symbols

$\mathcal{A}$	mapper
$\mathcal{A}_S$	symbolic mapper
$\mathcal{H}$	hypothesis (Mealy machine)
$\mathcal{M}$	Mealy machine
$\mathcal{M}_S$	symbolic Mealy Machine
H	set of states of hypothesis
I	set of (concrete) input symbols
O	set of (concrete) output symbols
Q	set of states of a Mealy machine
R	set of mapper states
T	set of event terms
V	set of variables
X	set of (abstract) input symbols
Y	set of (abstract) output symbols
a	(input or output) symbol
d	parameter value
e	term
h	state of hypothesis
h_0	initial state of hypothesis
i	(concrete) input symbol
j, k, l, m, n	index
o	(concrete) output symbol
p	parameter
q	state of Mealy machine
q_0	initial state of Mealy machine
r	state of mapper
r_0	initial state of mapper
s	sequence of output symbols
t	term
u	sequence of input symbols
v	variable
x	abstract input symbol
y	abstract output symbol

$\alpha_{\mathcal{A}}$	abstraction induced by $\mathcal{A}$
$\gamma_{\mathcal{A}}$	concretization induced by $\mathcal{A}$
δ	update function
ϵ	empty sequence
ε	event primitive
ξ	valuation
$\tau_{\mathcal{A}}$	observation abstraction function induced by $\mathcal{A}$
φ	formula
ψ	abstraction function
Δ	set of (symbolic) transitions
Θ	initial condition
Σ	event signature
Ψ	set of event abstractions
$\bot$	undefined value
$\rightarrow$	transition relation
$\Rightarrow$	transition relation extended to sequences
$\rightsquigarrow$	zip function as arrow
$\approx$	observation equivalence
$\leq$	implementation preorder / behavior inclusion
$\equiv$	syntactic equality